SICP in Python

wizardforcel

Published with GitBook



Table of Contents

SICP in Python

From: CS61A: Online Textbook

This book is derived from the classic textbook Structure and Interpretation of Computer Programs by Abelson, Sussman, and Sussman. John Denero originally modified if for Python for the Fall 2011 semester.

Chapter 1: Building Abstractions with Functions

Contents

1.1 Introduction

Computer science is a tremendously broad academic discipline. The areas of globally distributed systems, artificial intelligence, robotics, graphics, security, scientific computing, computer architecture, and dozens of emerging sub-fields each expand with new techniques and discoveries every year. The rapid progress of computer science has left few aspects of human life unaffected. Commerce, communication, science, art, leisure, and politics have all been reinvented as computational domains.

The tremendous productivity of computer science is only possible because it is built upon an elegant and powerful set of fundamental ideas. All computing begins with representing information, specifying logic to process it, and designing abstractions that manage the complexity of that logic. Mastering these fundamentals will require us to understand precisely how computers interpret computer programs and carry out computational processes.

These fundamental ideas have long been taught at Berkeley using the classic textbook *Structure and Interpretation of Computer Programs* (SICP) by Harold Abelson and Gerald Jay Sussman with Julie Sussman. These lecture notes borrow heavily from that textbook, which the original authors have kindly licensed for adaptation and reuse.

The embarkment of our intellectual journey requires no revision, nor should we expect that it ever will.

> We are about to study the idea of a *computational process*. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called data. The evolution of a process is directed by a pattern of rules called a program. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells. > > The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform. > > A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the sorcerer's apprentice, novice programmers must learn to understand and to anticipate the consequences of their conjuring. > > —Abelson and Sussman, SICP (1993)

1.1.1 Programming in Python

> A language isn’t something you learn so much as something you join. > > —Arika Okrent

In order to define computational processes, we need a programming language; preferably one many humans and a great variety of computers can all understand. In this course, we will learn the Python language.

Python is a widely used programming language that has recruited enthusiasts from many professions: web programmers, game engineers, scientists, academics, and even designers of new programming languages. When you learn Python, you join a million-person-strong community of developers. Developer communities are tremendously important institutions: members help each other solve problems, share their code and experiences, and collectively develop software and tools. Dedicated members often achieve celebrity and widespread esteem for their contributions. Perhaps someday you will be named among these elite Pythonistas.

The Python language itself is the product of a large volunteer community that prides itself on the diversity of its contributors. The language was conceived and first implemented by Guido van Rossum in the late 1980's. The first chapter of his Python 3 Tutorial explains why Python is so popular, among the many languages available today.

Python excels as an instructional language because, throughout its history, Python's developers have emphasized the human interpretability of Python code, reinforced by the Zen of Python guiding principles of beauty, simplicity, and readability. Python is particularly appropriate for this course because its broad set of features support a variety of different programming styles, which we will explore. While there is no single way to program in Python, there are a set of conventions shared across the developer community that facilitate the process of reading, understanding, and extending existing programs. Hence, Python's combination of great flexibility and accessibility allows students to explore many programming paradigms, and then apply their newly acquired knowledge to thousands of ongoing projects.

These notes maintain the spirit of SICP by introducing the features of Python in lock step with techniques for abstraction design and a rigorous model of computation. In addition, these notes provide a practical introduction to Python programming, including some advanced language features and illustrative examples. Learning Python will come naturally as you progress through the course.

However, Python is a rich language with many features and uses, and we consciously introduce them slowly as we layer on fundamental computer science concepts. For experienced students who want to inhale all of the details of the language quickly, we recommend reading Mark Pilgrim's book Dive Into Python 3, which is freely available online. The topics in that book differ substantially from the topics of this course, but the book contains very valuable practical information on using the Python language. Be forewarned: unlike these notes, Dive Into Python 3 assumes substantial programming experience.

The best way to get started programming in Python is to interact with the interpreter directly. This section describes how to install Python 3, initiate an interactive session with the interpreter, and start programming.

1.1.2 Installing Python 3

As with all great software, Python has many versions. This course will use the most recent stable version of Python 3 (currently Python 3.2). Many computers have older versions of Python installed already, but those will not suffice for this course. You should be able to use any computer for this course, but expect to install Python 3. Don't worry, Python is free.

Dive Into Python 3 has detailed installation instructions for all major platforms. These instructions mention Python 3.1 several times, but you're better off with Python 3.2 (although the differences are insignificant for this course). All instructional machines in the EECS department have Python 3.2 already installed.

1.1.3 Interactive Sessions

In an interactive Python session, you type some Python *code* after the *prompt*, `>>>` . The Python *interpreter* reads and evaluates what you type, carrying out your various commands.

There are several ways to start an interactive session, and they differ in their properties. Try them all to find out what you prefer. They all use exactly the same interpreter behind the scenes.

- The simplest and most common way is to run the Python 3 application. Type `python3` at a terminal prompt (Mac/Unix/Linux) or open the Python 3 application in Windows.
- A more user-friendly application for those learning the language is called Idle 3 (`idle3`). Idle colorizes your code (called syntax highlighting), pops up usage hints, and marks the source of some errors. Idle is always bundled with Python, so you have already installed it.
- The Emacs editor can run an interactive session inside one of its buffers. While slightly more challenging to learn, Emacs is a powerful and versatile editor for any programming language. Read the 61A Emacs Tutorial to get started. Many programmers who invest the time to learn Emacs never switch editors again.

In any case, if you see the Python prompt, `>>>` , then you have successfully started an interactive session. These notes depict example interactions using the prompt, followed by some input.

```
>>> 2 + 2
4
```

Controls: Each session keeps a history of what you have typed. To access that history, press `<Control>-P` (previous) and `<Control>-N` (next). `<Control>-D` exits a session, which discards this history.

1.1.4 First Example

> And, as imagination bodies forthThe forms of things to unknown, and the poet's penTurns them to shapes, and gives to airy nothingA local habitation and a name. > > —William Shakespeare, A Midsummer-Night's Dream

To give Python the introduction it deserves, we will begin with an example that uses several language features. In the next section, we will have to start from scratch and build up the language piece by piece. Think of this section as a sneak preview of powerful features to come.

Python has built-in support for a wide range of common programming activities, like manipulating text, displaying graphics, and communicating over the Internet. The import statement

```
>>> from urllib.request import urlopen
```

loads functionality for accessing data on the Internet. In particular, it makes available a function called `urlopen`, which can access the content at a uniform resource locator (URL), which is a location of something on the Internet.

Statements & Expressions. Python code consists of statements and expressions. Broadly, computer programs consist of instructions to either

1. Compute some value
2. Carry out some action

Statements typically describe actions. When the Python interpreter executes a statement, it carries out the corresponding action. On the other hand, expressions typically describe computations that yield values. When Python evaluates an expression, it computes its value. This chapter introduces several types of statements and expressions.

The assignment statement

```
>>> shakespeare = urlopen('http://inst.eecs.berkeley.edu/~cs61a/fa11/shakespeare.txt')
```

associates the name `shakespeare` with the value of the expression that follows. That expression applies the `urlopen` function to a URL that contains the complete text of William Shakespeare's 37 plays, all in a single text document.

Functions. Functions encapsulate logic that manipulates data. A web address is a piece of data, and the text of Shakespeare's plays is another. The process by which the former leads to the latter may be complex, but we can apply that process using only a simple expression because that complexity is tucked away within a function. Functions are the primary topic of this chapter.

Another assignment statement

```
>>> words = set(shakespeare.read().decode().split())
```

associates the name `words` to the set of all unique words that appear in Shakespeare's plays, all 33,721 of them. The chain of commands to `read`, `decode`, and `split`, each operate on an intermediate computational entity: data is read from the opened URL, that data is decoded into text, and that text is split into words. All of those words are placed in a `set`.

Objects. A set is a type of object, one that supports set operations like computing intersections and testing membership. An object seamlessly bundles together data and the logic that manipulates that data, in a way that hides the complexity of both. Objects are the primary topic of Chapter 2.

The expression

```
>>> {w for w in words if len(w) >= 5 and w[::-1] in words}
{'madam', 'stink', 'leets', 'rever', 'drawer', 'stops', 'sessa',
'repaid', 'speed', 'redder', 'devil', 'minim', 'spots', 'asses',
'refer', 'lived', 'keels', 'diaper', 'sleek', 'steel', 'leper',
'level', 'deeps', 'repel', 'reward', 'knits'}
```

is a compound expression that evaluates to the set of Shakespearian words that appear both forward and in reverse. The cryptic notation `w[::-1]` enumerates each letter in a word, but the `-1` says to step backwards (`::` here means that the positions of the first and last characters to enumerate are defaulted.) When you enter an expression in an interactive session, Python prints its value on the following line, as shown.

Interpreters. Evaluating compound expressions requires a precise procedure that interprets code in a predictable way. A program that implements such a procedure, evaluating compound expressions and statements, is called an interpreter. The design and implementation of interpreters is the primary topic of Chapter 3.

When compared with other computer programs, interpreters for programming languages are unique in their generality. Python was not designed with Shakespeare or palindromes in mind. However, its great flexibility allowed us to process a large amount of text with only a few lines of code.

In the end, we will find that all of these core concepts are closely related: functions are objects, objects are functions, and interpreters are instances of both. However, developing a clear understanding of each of these concepts and their role in organizing code is critical to mastering the art of programming.

1.1.5 Practical Guidance: Errors

Python is waiting for your command. You are encouraged to experiment with the language, even though you may not yet know its full vocabulary and structure. However, be prepared for errors. While computers are tremendously fast and flexible, they are also extremely rigid. The nature of computers is described in Stanford's introductory course as

> The fundamental equation of computers is: `computer = powerful + stupid` > > Computers are very powerful, looking at volumes of data very quickly. Computers can perform billions of operations per second, where each operation is pretty simple. > > Computers are also shockingly stupid and fragile. The operations that they can do are extremely rigid, simple, and mechanical. The computer lacks anything like real insight .. it's nothing like the HAL 9000 from the movies. If nothing else, you should not be intimidated by the computer as if it's some sort of brain. It's very mechanical underneath it all. > > Programming is about a person using their real insight to build something useful, constructed out of these teeny, simple little operations that the computer can do. > > —Francisco Cai and Nick Parlante, Stanford CS101

The rigidity of computers will immediately become apparent as you experiment with the Python interpreter: even the smallest spelling and formatting changes will cause unexpected outputs and errors.

Learning to interpret errors and diagnose the cause of unexpected errors is called *debugging*. Some guiding principles of debugging are:

1. **Test incrementally**: Every well-written program is composed of small, modular components that can be tested individually. Test everything you write as soon as possible to catch errors early and gain confidence in your components.
2. **Isolate errors**: An error in the output of a compound program, expression, or statement can typically be attributed to a particular modular component. When trying to diagnose a problem, trace the error to the smallest fragment of code you can before trying to correct it.

3. **Check your assumptions**: Interpreters do carry out your instructions to the letter --- no more and no less. Their output is unexpected when the behavior of some code does not match what the programmer believes (or assumes) that behavior to be. Know your assumptions, then focus your debugging effort on verifying that your assumptions actually hold.
4. **Consult others**: You are not alone! If you don't understand an error message, ask a friend, instructor, or search engine. If you have isolated an error, but can't figure out how to correct it, ask someone else to take a look. A lot of valuable programming knowledge is shared in the context of team problem solving.

Incremental testing, modular design, precise assumptions, and teamwork are themes that persist throughout this course. Hopefully, they will also persist throughout your computer science career.

1.2 The Elements of Programming

A programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes. Programs serve to communicate those ideas among the members of a programming community. Thus, programs must be written for people to read, and only incidentally for machines to execute.

When we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

- **primitive expressions and statements**, which represent the simplest building blocks that the language provides,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

In programming, we deal with two kinds of elements: functions and data. (Soon we will discover that they are really not so distinct.) Informally, data is stuff that we want to manipulate, and functions describe the rules for manipulating the data. Thus, any powerful programming language should be able to describe primitive data and primitive functions and should have methods for combining and abstracting both functions and data.

1.2.1 Expressions

Having experimented with the full Python interpreter, we now must start anew, methodically developing the Python language piece by piece. Be patient if the examples seem simplistic --- more exciting material is soon to come.

We begin with primitive expressions. One kind of primitive expression is a number. More precisely, the expression that you type consists of the numerals that represent the number in base 10.

```
>>> 42
42
```

Expressions representing numbers may be combined with mathematical operators to form a compound expression, which the interpreter will evaluate:

```
>>> -1 - -1
0
>>> 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
0.9921875
```

These mathematical expressions use *infix* notation, where the *operator* (e.g., `+`, `-`, `*`, or `/`) appears in between the *operands* (numbers). Python includes many ways to form compound expressions. Rather than attempt to enumerate them all immediately, we will introduce new expression forms as we go, along with the language features that they support.

1.2.2 Call Expressions

The most important kind of compound expression is a *call expression*, which applies a function to some arguments. Recall from algebra that the mathematical notion of a function is a mapping from some input arguments to an output value. For instance, the `max` function maps its inputs to a single output, which is the largest of the inputs. A function in Python is more than just an input-output mapping; it describes a computational process. However, the way in which Python expresses function application is the same as in mathematics.

```
>>> max(7.5, 9.5)
9.5
```

This call expression has subexpressions: the operator precedes parentheses, which enclose a comma-delimited list of operands. The operator must be a function. The operands can be any values; in this case they are numbers. When this call expression is evaluated, we say that the function `max` is *called* with arguments 7.5 and 9.5, and *returns* a value of 9.5.

The order of the arguments in a call expression matters. For instance, the function `pow` raises its first argument to the power of its second argument.

```
>>> pow(100, 2)
10000
>>> pow(2, 100)
1267650600228229401496703205376
```

Function notation has several advantages over the mathematical convention of infix notation. First, functions may take an arbitrary number of arguments:

```
>>> max(1, -2, 3, -4)
3
```

No ambiguity can arise, because the function name always precedes its arguments.

Second, function notation extends in a straightforward way to *nested* expressions, where the elements are themselves compound expressions. In nested call expressions, unlike compound infix expressions, the structure of the nesting is entirely explicit in the parentheses.

```
>>> max(min(1, -2), min(pow(3, 5), -4))
-2
```

There is no limit (in principle) to the depth of such nesting and to the overall complexity of the expressions that the Python interpreter can evaluate. However, humans quickly get confused by multi-level nesting. An important role for you as a programmer is to structure expressions so that they remain interpretable by yourself, your programming partners, and others who may read your code in the future.

Finally, mathematical notation has a great variety of forms: multiplication appears between terms, exponents appear as superscripts, division as a horizontal bar, and a square root as a roof with slanted siding. Some of this notation is very hard to type! However, all of this complexity can be unified via the notation of call expressions. While Python supports common mathematical operators using infix notation (like `+` and `-`), any operator can be expressed as a function with a name.

1.2.3 Importing Library Functions

Python defines a very large number of functions, including the operator functions mentioned in the preceding section, but does not make their names available by default, so as to avoid complete chaos. Instead, it organizes the functions and other quantities that it knows about into modules, which together comprise the Python Library. To use these elements, one imports them. For example, the `math` module provides a variety of familiar mathematical functions:

```
>>> from math import sqrt, exp
>>> sqrt(256)
16.0
>>> exp(1)
2.718281828459045
```

and the `operator` module provides access to functions corresponding to infix operators:

```
>>> from operator import add, sub, mul
>>> add(14, 28)
42
>>> sub(100, mul(7, add(8, 4)))
16
```

An `import` statement designates a module name (e.g., `operator` or `math`), and then lists the named attributes of that module to import (e.g., `sqrt` or `exp`).

The Python 3 Library Docs list the functions defined by each module, such as the math module. However, this documentation is written for developers who know the whole language well. For now, you may find that experimenting with a function tells you more about its behavior than reading the documemtation. As you become familiar with the Python language and vocabulary, this documentation will become a valuable reference source.

1.2.4 Names and the Environment

A critical aspect of a programming language is the means it provides for using names to refer to computational objects. If a value has been given a name, we say that the name *binds* to the value.

In Python, we can establish new bindings using the assignment statement, which contains a name to the left of `=` and a value to the right:

```
>>> radius = 10
>>> radius
10
>>> 2 * radius
20
```

Names are also bound via `import` statements.

```
>>> from math import pi
>>> pi * 71 / 223
1.0002380197528042
```

We can also assign multiple values to multiple names in a single statement, where names and expressions are separated by commas.

```
>>> area, circumference = pi * radius * radius, 2 * pi * radius
>>> area
314.1592653589793
>>> circumference
62.83185307179586
```

The `=` symbol is called the *assignment* operator in Python (and many other languages). Assignment is Python's simplest means of *abstraction*, for it allows us to use simple names to refer to the results of compound operations, such as the `area` computed above. In this way, complex programs are constructed by building, step by step, computational objects of increasing complexity.

The possibility of binding names to values and later retrieving those values by name means that the interpreter must maintain some sort of memory that keeps track of the names, values, and bindings. This memory is called an *environment*.

Names can also be bound to functions. For instance, the name `max` is bound to the max function we have been using. Functions, unlike numbers, are tricky to render as text, so Python prints an identifying description instead, when asked to print a function:

```
>>> max
<built-in function max>
```

We can use assignment statements to give new names to existing functions.

```
>>> f = max
>>> f
<built-in function max>
>>> f(3, 4)
4
```

And successive assignment statements can rebind a name to a new value.

```
>>> f = 2
>>> f
2
```

In Python, the names bound via assignment are often called *variable names* because they can be bound to a variety of different values in the course of executing a program.

1.2.5 Evaluating Nested Expressions

One of our goals in this chapter is to isolate issues about thinking procedurally. As a case in point, let us consider that, in evaluating nested call expressions, the interpreter is itself following a procedure.

To evaluate a call expression, Python will do the following:

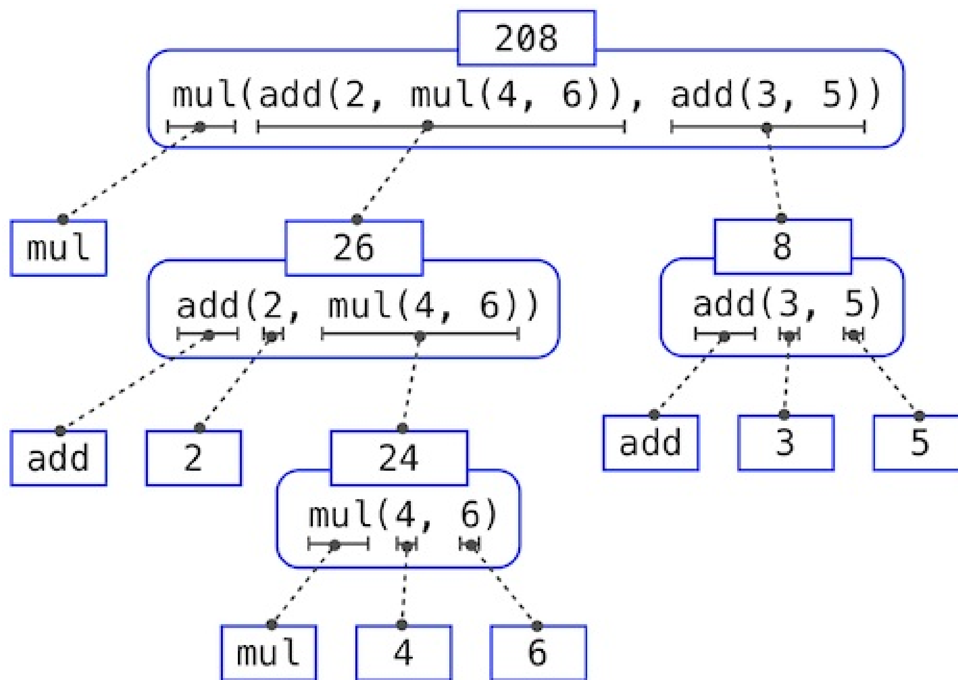
1. Evaluate the operator and operand subexpressions, then
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

Even this simple procedure illustrates some important points about processes in general. The first step dictates that in order to accomplish the evaluation process for a call expression we must first evaluate other expressions. Thus, the evaluation procedure is *recursive* in nature; that is, it includes, as one of its steps, the need to invoke the rule itself.

For example, evaluating

```
>>> mul(add(2, mul(4, 6)), add(3, 5))
208
```

requires that this evaluation procedure be applied four times. If we draw each expression that we evaluate, we can visualize the hierarchical structure of this process.



This illustration is called an *expression tree*. In computer science, trees grow from the top down. The objects at each point in a tree are called nodes; in this case, they are expressions paired with their values.

Evaluating its root, the full expression, requires first evaluating the branches that are its subexpressions. The leaf expressions (that is, nodes with no branches stemming from them) represent either functions or numbers. The interior nodes have two parts: the call expression to which our evaluation rule is applied, and the result of that expression. Viewing evaluation in terms of this tree, we can imagine that the values of the operands percolate upward, starting from the terminal nodes and then combining at higher and higher levels.

Next, observe that the repeated application of the first step brings us to the point where we need to evaluate, not call expressions, but primitive expressions such as numerals (e.g., `2`) and names (e.g., `add`). We take care of the primitive cases by stipulating that

- A numeral evaluates to the number it names,
- A name evaluates to the value associated with that name in the current environment.

Notice the important role of an environment in determining the meaning of the symbols in expressions. In Python, it is meaningless to speak of the value of an expression such as

```
>>> add(x, 1)
```

without specifying any information about the environment that would provide a meaning for the name `x` (or even for the name `add`). Environments provide the context in which evaluation takes place, which plays an important role in our understanding of program execution.

This evaluation procedure does not suffice to evaluate all Python code, only call expressions, numerals, and names. For instance, it does not handle assignment statements. Executing

```
>>> x = 3
```

does not return a value nor evaluate a function on some arguments, since the purpose of assignment is instead to bind a name to a value. In general, statements are not evaluated but *executed*; they do not produce a value but instead make some change. Each type of statement or expression has its own evaluation or execution procedure, which we will introduce incrementally as we proceed.

A pedantic note: when we say that "a numeral evaluates to a number," we actually mean that the Python interpreter evaluates a numeral to a number. It is the interpreter which endows meaning to the programming language. Given that the interpreter is a fixed program that always behaves consistently, we can loosely say that numerals (and expressions) themselves evaluate to values in the context of Python programs.

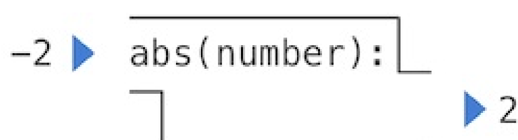
1.2.6 Function Diagrams

As we continue to develop a formal model of evaluation, we will find that diagramming the internal state of the interpreter helps us track the progress of our evaluation procedure. An essential part of these diagrams is a representation of a function.

Pure functions. Functions have some input (their arguments) and return some output (the result of applying them). The built-in function

```
>>> abs(-2)
2
```

can be depicted as a small machine that takes input and produces output.

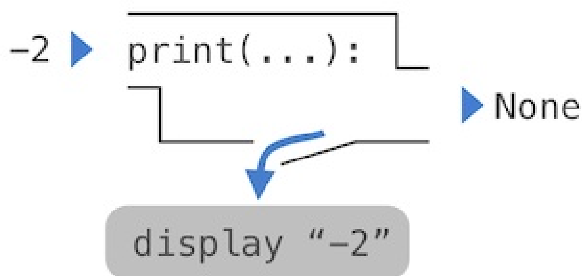


The function `abs` is *pure*. Pure functions have the property that applying them has no effects beyond returning a value.

Non-pure functions. In addition to returning a value, applying a non-pure function can generate *side effects*, which make some change to the state of the interpreter or computer. A common side effect is to generate additional output beyond the return value, using the `print` function.

```
>>> print(-2)
-2
>>> print(1, 2, 3)
1 2 3
```

While `print` and `abs` may appear to be similar in these examples, they work in fundamentally different ways. The value that `print` returns is always `None`, a special Python value that represents nothing. The interactive Python interpreter does not automatically print the value `None`. In the case of `print`, the function itself is printing output as a side effect of being called.



A nested expression of calls to `print` highlights the non-pure character of the function.

```
>>> print(print(1), print(2))
1
2
None None
```

If you find this output to be unexpected, draw an expression tree to clarify why evaluating this expression produces this peculiar output.

Be careful with `print`! The fact that it returns `None` means that it *should not* be the expression in an assignment statement.

```
>>> two = print(2)
2
>>> print(two)
None
```

Signatures. Functions differ in the number of arguments that they are allowed to take. To track these requirements, we draw each function in a way that shows the function name and names of its arguments. The function `abs` takes only one argument called `number`; providing more or fewer will result in an error. The function `print` can take an arbitrary number of arguments, hence its rendering as `print(...)`. A description of the arguments that a function can take is called the function's *signature*.

1.3 Defining New Functions

We have identified in Python some of the elements that must appear in any powerful programming language:

1. Numbers and arithmetic operations are built-in data and functions.
2. Nested function application provides a means of combining operations.
3. Binding names to values provides a limited means of abstraction.

Now we will learn about *function definitions*, a much more powerful abstraction technique by which a name can be bound to compound operation, which can then be referred to as a unit.

We begin by examining how to express the idea of "squaring." We might say, "To square something, multiply it by itself." This is expressed in Python as

```
>>> def square(x):
        return mul(x, x)
```

which defines a new function that has been given the name `square`. This user-defined function is not built into the interpreter. It represents the compound operation of multiplying something by itself. The `x` in this definition is called a *formal parameter*, which provides a name for the thing to be multiplied. The definition creates this user-defined function and associates it with the name `square`.

Function definitions consist of a `def` statement that indicates a `<name>` and a list of named `<formal parameters>`, then a `return` statement, called the function body, that specifies the `<return expression>` of the function, which is an expression to be evaluated whenever the function is applied.

> `def <name>(<formal parameters>):` > > `return <return expression>`

The second line *must* be indented! Convention dictates that we indent with four spaces, rather than a tab. The return expression is not evaluated right away; it is stored as part of the newly defined function and evaluated only when the function is eventually applied. (Soon, we will see that the indented region can span multiple lines.)

Having defined `square`, we can apply it with a call expression:

```
>>> square(21)
441
>>> square(add(2, 5))
49
>>> square(square(3))
81
```

We can also use `square` as a building block in defining other functions. For example, we can easily define a function `sum_squares` that, given any two numbers as arguments, returns the sum of their squares:

```
>>> def sum_squares(x, y):
        return add(square(x), square(y))
```

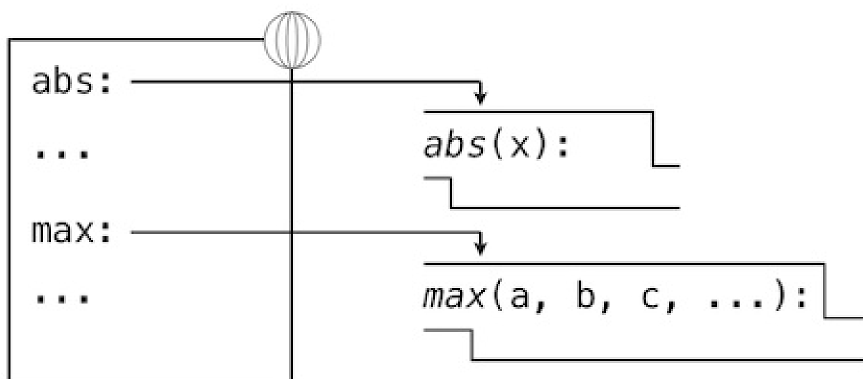
```
>>> sum_squares(3, 4)
25
```

User-defined functions are used in exactly the same way as built-in functions. Indeed, one cannot tell from the definition of `sum_squares` whether `square` is built into the interpreter, imported from a module, or defined by the user.

1.3.1 Environments

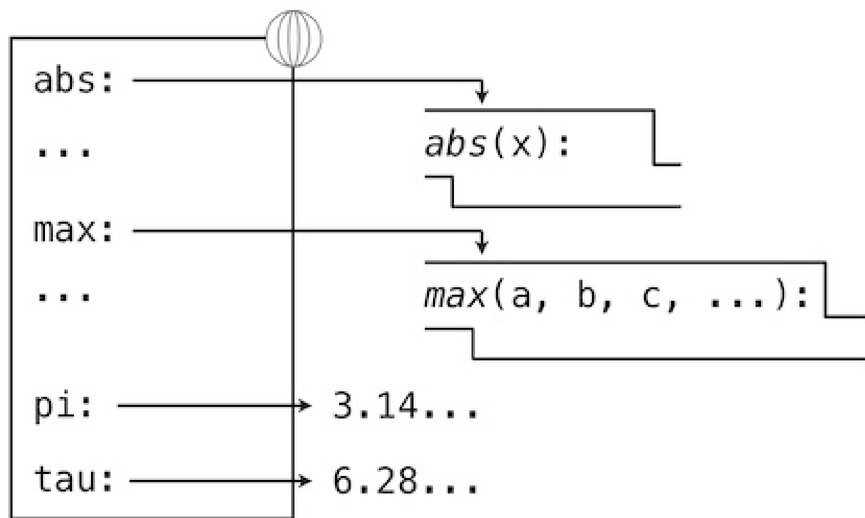
Our subset of Python is now complex enough that the meaning of programs is non-obvious. What if a formal parameter has the same name as a built-in function? Can two functions share names without confusion? To resolve such questions, we must describe environments in more detail.

An environment in which an expression is evaluated consists of a sequence of *frames*, depicted as boxes. Each frame contains *bindings*, which associate a name with its corresponding value. There is a single *global* frame that contains name bindings for all built-in functions (only `abs` and `max` are shown). We indicate the global frame with a globe symbol.

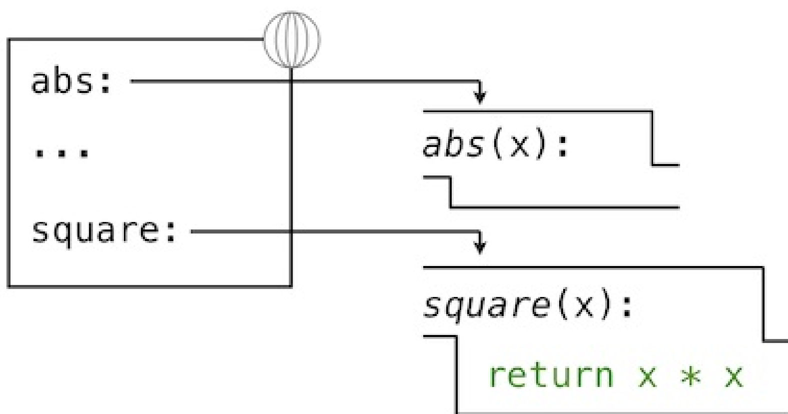


Assignment and import statements add entries to the first frame of the current environment. So far, our environment consists only of the global frame.

```
>>> from math import pi
>>> tau = 2 * pi
```



A `def` statement also binds a name to the function created by the definition. The resulting environment after defining `square` appears below:



These *environment diagrams* show the bindings of the current environment, along with the values (which are not part of any frame) to which names are bound. Notice that the name of a function is repeated, once in the frame, and once as part of the function itself. This repetition is intentional: many different names may refer to the same function, but that function itself has only one intrinsic name. However, looking up the value for a name in an environment only inspects name bindings. The intrinsic name of a function **does not** play a role in looking up names. In the example we saw earlier,

```
>>> f = max
>>> f
<built-in function max>
```

The name *max* is the intrinsic name of the function, and that's what you see printed as the value for `f`. In addition, both the names `max` and `f` are bound to that same function in the global environment.

As we proceed to introduce additional features of Python, we will have to extend these diagrams. Every time we do, we will list the new features that our diagrams can express.

New environment Features: Assignment and user-defined function definition.

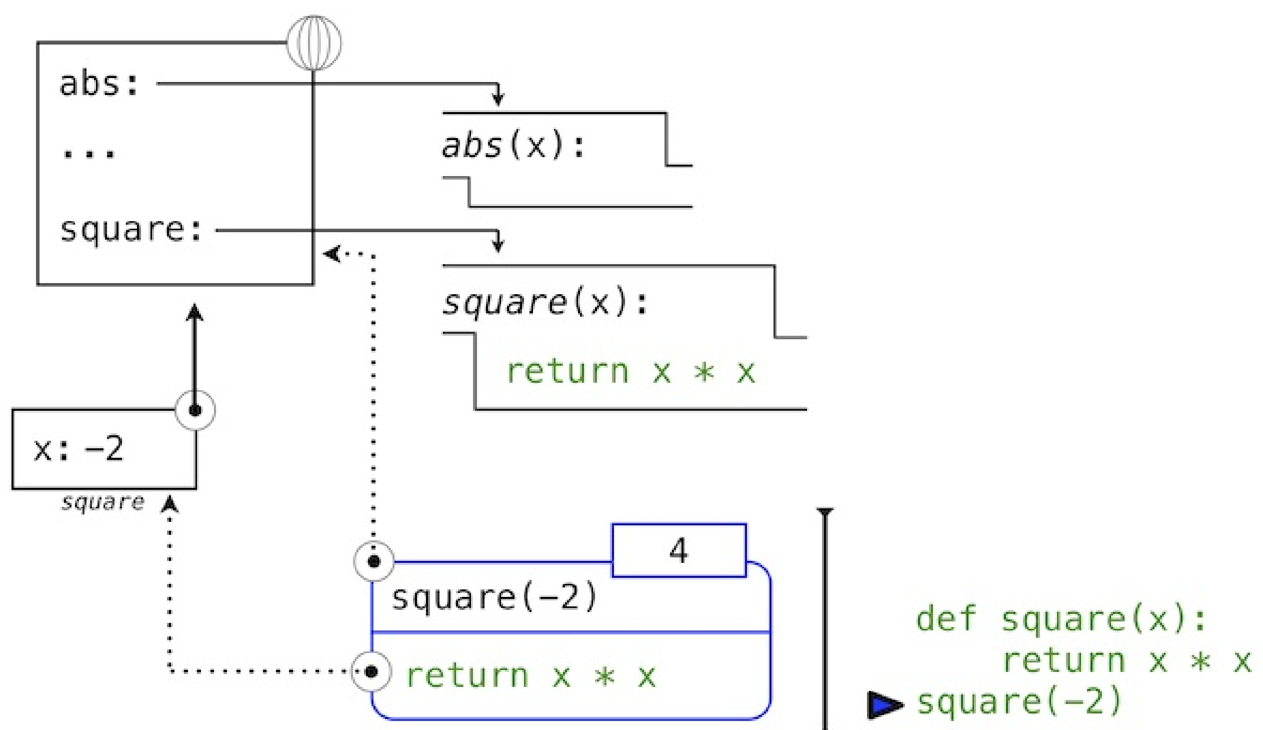
1.3.2 Calling User-Defined Functions

To evaluate a call expression whose operator names a user-defined function, the Python interpreter follows a process similar to the one for evaluating expressions with a built-in operator function. That is, the interpreter evaluates the operand expressions, and then applies the named function to the resulting arguments.

The act of applying a user-defined function introduces a second *local* frame, which is only accessible to that function. To apply a user-defined function to some arguments:

1. Bind the arguments to the names of the function's formal parameters in a new *local* frame.
2. Evaluate the body of the function in the environment beginning at that frame and ending at the global frame.

The environment in which the body is evaluated consists of two frames: first the local frame that contains argument bindings, then the global frame that contains everything else. Each instance of a function application has its own independent local frame.



This figure includes two different aspects of the Python interpreter: the current environment, and a part of the expression tree related to the current line of code being evaluated. We have depicted the evaluation of a call expression that has a user-defined function (in blue) as a two-part rounded rectangle. Dotted arrows indicate which environment is used to evaluate the expression in each part.

- The top half shows the call expression being evaluated. This call expression is not internal to any function, so it is evaluated in the global environment. Thus, any names within it (such as `square`) are looked up in the global frame.
- The bottom half shows the body of the `square` function. Its return expression is evaluated in the new environment introduced by step 1 above, which binds the name of `square`'s formal parameter `x` to the value of its argument, `-2`.

The order of frames in an environment affects the value returned by looking up a name in an expression. We stated previously that a name is evaluated to the value associated with that name in the current environment. We can now be more precise:

- A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Our conceptual framework of environments, names, and functions constitutes a *model of evaluation*; while some mechanical details are still unspecified (e.g., how a binding is implemented), our model does precisely and correctly describe how the interpreter evaluates call expressions. In Chapter 3 we shall see how this model can serve as a blueprint for implementing a working interpreter for a programming language.

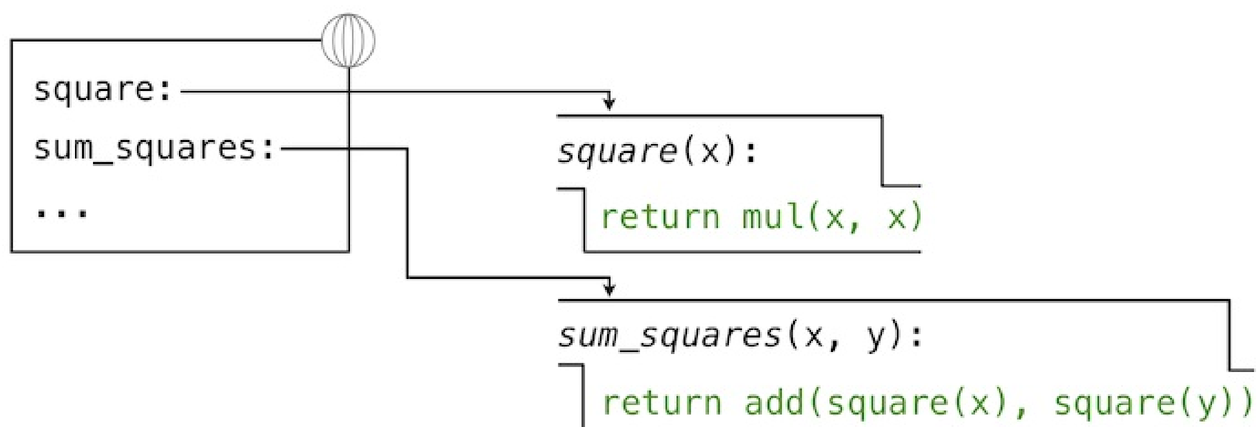
New environment Feature: Function application.

1.3.3 Example: Calling a User-Defined Function

Let us again consider our two simple definitions:

```
>>> from operator import add, mul
>>> def square(x):
        return mul(x, x)
```

```
>>> def sum_squares(x, y):
        return add(square(x), square(y))
```

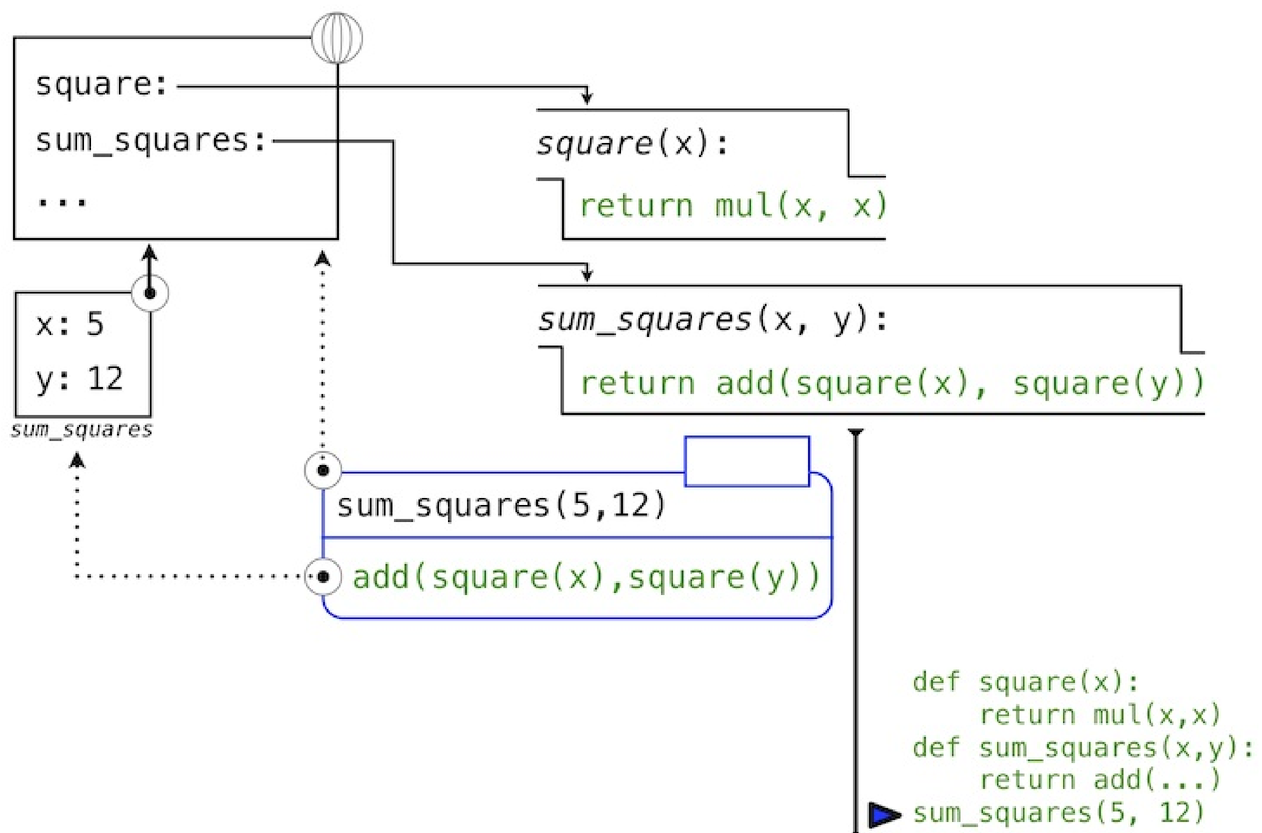


And the process that evaluates the following call expression:

```
>>> sum_squares(5, 12)
169
```

Python first evaluates the name `sum_squares`, which is bound to a user-defined function in the global frame. The primitive numeric expressions 5 and 12 evaluate to the numbers they represent.

Next, Python applies `sum_squares`, which introduces a local frame that binds x to 5 and y to 12.



In this diagram, the local frame points to its successor, the global frame. All local frames must point to a predecessor, and these links define the sequence of frames that is the current environment.

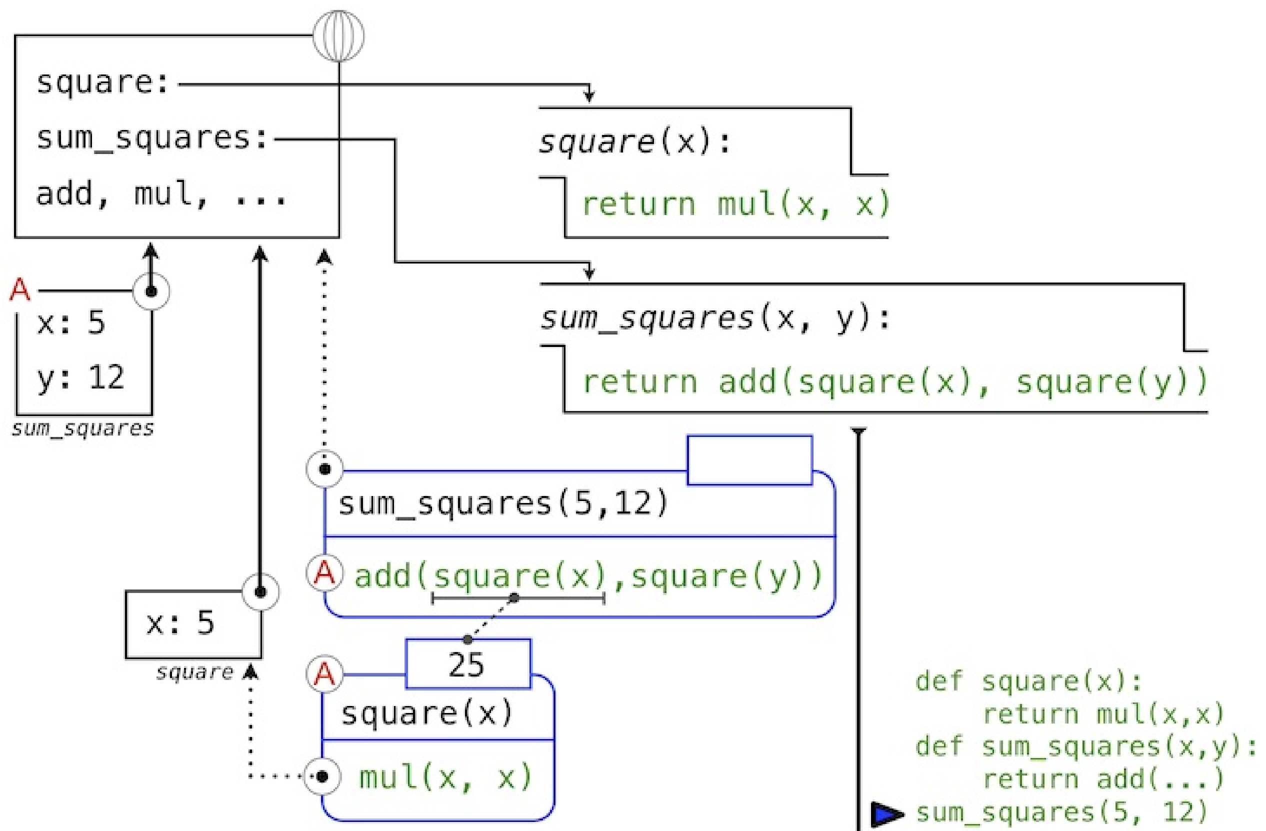
The body of `sum_squares` contains this call expression:

```
   add      (  square(x)  ,  square(y)  )
 ________      _________     _________
"operator"   "operand 0"   "operand 1"
```

All three subexpressions are evalauted in the current environment, which begins with the frame labeled *sum_squares*. The operator subexpression `add` is a name found in the global frame, bound to the built-in function for addition. The two operand subexpressions must be

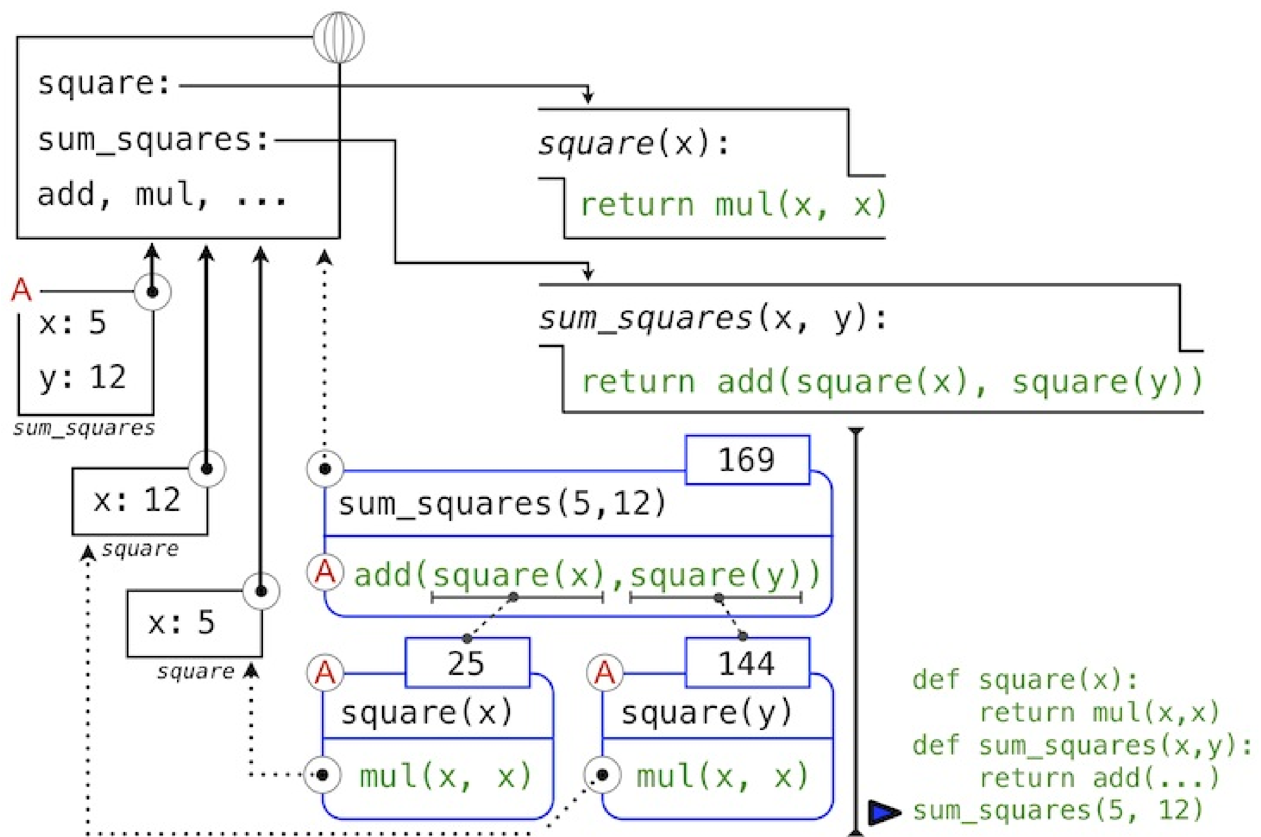
evaluated in turn, before addition is applied. Both operands are evaluated in the current environment beginning with the frame labeled `sum_squares`. In the following environment diagrams, we will call this frame `A` and replace arrows pointing to this frame with the label `A` as well.

In `operand 0`, `square` names a user-defined function in the global frame, while `x` names the number 5 in the local frame. Python applies `square` to 5 by introducing yet another local frame that binds x to 5.



Using this local frame, the body expression `mul(x, x)` evaluates to 25.

Our evaluation procedure now turns to `operand 1`, for which `y` names the number 12. Python evaluates the body of `square` again, this time introducing yet another local environment frame that binds `x` to 12. Hence, `operand 1` evaluates to 144.



Finally, applying addition to the arguments 25 and 144 yields a final value for the body of `sum_squares` : 169.

This figure, while complex, serves to illustrate many of the fundamental ideas we have developed so far. Names are bound to values, which spread across many local frames that all precede a single global frame that contains shared names. Expressions are tree-structured, and the environment must be augmented each time a subexpression contains a call to a user-defined function.

All of this machinery exists to ensure that names resolve to the correct values at the correct points in the expression tree. This example illustrates why our model requires the complexity that we have introduced. All three local frames contain a binding for the name `x` , but that name is bound to different values in different frames. Local frames keep these names separate.

1.3.4 Local Names

One detail of a function's implementation that should not affect the function's behavior is the implementer's choice of names for the function's formal parameters. Thus, the following functions should provide the same behavior:

```
>>> def square(x):
        return mul(x, x)
>>> def square(y):
        return mul(y, y)
```

This principle -- that the meaning of a function should be independent of the parameter names chosen by its author -- has important consequences for programming languages. The simplest consequence is that the parameter names of a function must remain local to the body of the function.

If the parameters were not local to the bodies of their respective functions, then the parameter `x` in `square` could be confused with the parameter `x` in `sum_squares`. Critically, this is not the case: the binding for `x` in different local frames are unrelated. Our model of computation is carefully designed to ensure this independence.

We say that the *scope* of a local name is limited to the body of the user-defined function that defines it. When a name is no longer accessible, it is out of scope. This scoping behavior isn't a new fact about our model; it is a consequence of the way environments work.

1.3.5 Practical Guidance: Choosing Names

The interchangeabily of names does not imply that formal parameter names do not matter at all. To the contrary, well-chosen function and parameter names are essential for the human interpretability of function definitions!

The following guidelines are adapted from the style guide for Python code, which serves as a guide for all (non-rebellious) Python programmers. A shared set of conventions smooths communication among members of a programming community. As a side effect of following these conventions, you will find that your code becomes more internally consistent.

1. Function names should be lowercase, with words separated by underscores. Descriptive names are encouraged.
2. Function names typically evoke operations applied to arguments by the interpreter (e.g., `print`, `add`, `square`) or the name of the quantity that results (e.g., `max`, `abs`, `sum`).
3. Parameter names should be lowercase, with words separated by underscores. Single-word names are preferred.
4. Parameter names should evoke the role of the parameter in the function, not just the type of value that is allowed.
5. Single letter parameter names are acceptable when their role is obvious, but never use "l" (lowercase ell), "O" (capital oh), or "I" (capital i) to avoid confusion with numerals.

Review these guidelines periodically as you write programs, and soon your names will be delightfully Pythonic.

1.3.6 Functions as Abstractions

Though it is very simple, `sum_squares` exemplifies the most powerful property of user-defined functions. The function `sum_squares` is defined in terms of the function `square`, but relies only on the relationship that `square` defines between its input arguments and its output values.

We can write `sum_squares` without concerning ourselves with *how* to square a number. The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as `sum_squares` is concerned, `square` is not a particular function body, but rather an abstraction of a function, a so-called functional abstraction. At this level of abstraction, any function that computes the square is equally good.

Thus, considering only the values they return, the following two functions for squaring a number should be indistinguishable. Each takes a numerical argument and produces the square of that number as the value.

```
>>> def square(x):
        return mul(x, x)
>>> def square(x):
        return mul(x, x-1) + x
```

In other words, a function definition should be able to suppress details. The users of the function may not have written the function themselves, but may have obtained it from another programmer as a "black box". A user should not need to know how the function is implemented in order to use it. The Python Library has this property. Many developers use the functions defined there, but few ever inspect their implementation. In fact, many implementations of Python Library functions are not written in Python at all, but instead in the C language.

1.3.7 Operators

Mathematical operators (like + and -) provided our first example of a method of combination, but we have yet to define an evaluation procedure for expressions that contain these operators.

Python expressions with infix operators each have their own evaluation procedures, but you can often think of them as short-hand for call expressions. When you see

```
>>> 2 + 3
5
```

simply consider it to be short-hand for

```
>>> add(2, 3)
5
```

Infix notation can be nested, just like call expressions. Python applies the normal mathematical rules of operator precedence, which dictate how to interpret a compound expression with multiple operators.

```
>>> 2 + 3 * 4 + 5
19
```

evaluates to the same result as

```
>>> add(add(2, mul(3, 4)) , 5)
19
```

The nesting in the call expression is more explicit than the operator version. Python also allows subexpression grouping with parentheses, to override the normal precedence rules or make the nested structure of an expression more explicit.

```
>>> (2 + 3) * (4 + 5)
45
```

evaluates to the same result as

```
>>> mul(add(2, 3), add(4, 5))
45
```

You should feel free to use these operators and parentheses in your programs. Idiomatic Python prefers operators over call expressions for simple mathematical operations.

1.4 Practical Guidance: The Art of the Function

Functions are an essential ingredient of all programs, large and small, and serve as our primary medium to express computational processes in a programming language. So far, we have discussed the formal properties of functions and how they are applied. We now turn to the topic of what makes a good function. Fundamentally, the qualities of good functions all reinforce the idea that functions are abstractions.

- Each function should have exactly one job. That job should be identifiable with a short name and characterizable in a single line of text. Functions that perform multiple jobs in sequence should be divided into multiple functions.
- *Don't repeat yourself* is a central tenet of software engineering. The so-called DRY principle states that multiple fragments of code should not describe redundant logic. Instead, that logic should be implemented once, given a name, and applied multiple times. If you find yourself copying and pasting a block of code, you have probably found an opportunity for functional abstraction.
- Functions should be defined generally. Squaring is not in the Python Library precisely because it is a special case of the `pow` function, which raises numbers to arbitrary powers.

These guidelines improve the readability of code, reduce the number of errors, and often minimize the total amount of code written. Decomposing a complex task into concise functions is a skill that takes experience to master. Fortunately, Python provides several features to support your efforts.

1.4.1 Docstrings

A function definition will often include documentation describing the function, called a *docstring*, which must be indented along with the function body. Docstrings are conventionally triple quoted. The first line describes the job of the function in one line. The following lines can describe arguments and clarify the behavior of the function:

```
>>> def pressure(v, t, n):
        """Compute the pressure in pascals of an ideal gas.

        Applies the ideal gas law: http://en.wikipedia.org/wiki/Ideal_gas_law

        v -- volume of gas, in cubic meters
        t -- absolute temperature in degrees kelvin
        n -- particles of gas
        """
        k = 1.38e-23  # Boltzmann's constant
        return n * k * t / v
```

When you call `help` with the name of a function as an argument, you see its docstring (type `q` to quit Python help).

```
>>> help(pressure)
```

When writing Python programs, include docstrings for all but the simplest functions. Remember, code is written only once, but often read many times. The Python docs include docstring guidelines that maintain consistency across different Python projects.

1.4.2 Default Argument Values

A consequence of defining general functions is the introduction of additional arguments. Functions with many arguments can be awkward to call and difficult to read.

In Python, we can provide default values for the arguments of a function. When calling that function, arguments with default values are optional. If they are not provided, then the default value is bound to the formal parameter name instead. For instance, if an application commonly computes pressure for one mole of particles, this value can be provided as a default:

```
>>> k_b=1.38e-23  # Boltzmann's constant
>>> def pressure(v, t, n=6.022e23):
        """Compute the pressure in pascals of an ideal gas.

        v -- volume of gas, in cubic meters
        t -- absolute temperature in degrees kelvin
        n -- particles of gas (default: one mole)
        """
        return n * k_b * t / v
```

```
>>> pressure(1, 273.15)
2269.974834
```

Here, `pressure` is defined to take three arguments, but only two are provided in the call expression that follows. In this case, the value for `n` is taken from the `def` statement defaults (which looks like an assignment to `n`, although as this discussion suggests, it is more of a conditional assignment.)

As a guideline, most data values used in a function's body should be expressed as default values to named arguments, so that they are easy to inspect and can be changed by the function caller. Some values that never change, like the fundamental constant `k_b`, can be defined in the global frame.

1.5 Control

The expressive power of the functions that we can define at this point is very limited, because we have not introduced a way to make tests and to perform different operations depending on the result of a test. *Control statements* will give us this capacity. Control statements differ fundamentally from the expressions that we have studied so far. They deviate from the strict evaluation of subexpressions from left to write, and get their name from the fact that they control what the interpreter should do next, possibly based on the values of expressions.

1.5.1 Statements

So far, we have primarily considered how to evaluate expressions. However, we have seen three kinds of statements: assignment, `def`, and `return` statements. These lines of Python code are not themselves expressions, although they all contain expressions as components.

To emphasize that the value of a statement is irrelevant (or nonexistant), we describe statements as being *executed* rather than evaluated. Each statement describes some change to the interpreter state, and executing a statement applies that change. As we have seen for `return` and assignment statements, executing statements can involve evaluating subexpressions contained within them.

Expressions can also be executed as statements, in which case they are evaluated, but their value is discarded. Executing a pure function has no effect, but executing a non-pure function can cause effects as a consequence of function application.

Consider, for instance,

```
>>> def square(x):
        mul(x, x) # Watch out! This call doesn't return a value.
```

This is valid Python, but probably not what was intended. The body of the function consists of an expression. An expression by itself is a valid statement, but the effect of the statement is that the `mul` function is called, and the result is discarded. If you want to do something with the result of an expression, you need to say so: you might store it with an assignment statement, or return it with a return statement:

```
>>> def square(x):
        return mul(x, x)
```

Sometimes it does make sense to have a function whose body is an expression, when a non-pure function like `print` is called.

```
>>> def print_square(x):
        print(square(x))
```

At its highest level, the Python interpreter's job is to execute programs, composed of statements. However, much of the interesting work of computation comes from evaluating expressions. Statements govern the relationship among different expressions in a program and what happens to their results.

1.5.2 Compound Statements

In general, Python code is a sequence of statements. A simple statement is a single line that doesn't end in a colon. A compound statement is so called because it is composed of other statements (simple and compound). Compound statements typically span multiple lines and start with a one-line header ending in a colon, which identifies the type of statement. Together, a header and an indented suite of statements is called a clause. A compound statement consists of one or more clauses:

```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

We can understand the statements we have already introduced in these terms.

- Expressions, return statements, and assignment statements are simple statements.
- A `def` statement is a compound statement. The suite that follows the `def` header defines the function body.

Specialized evaluation rules for each kind of header dictate when and if the statements in its suite are executed. We say that the header controls its suite. For example, in the case of `def` statements, we saw that the return expression is not evaluated immediately, but instead stored for later use when the defined function is eventually applied.

We can also understand multi-line programs now.

- To execute a sequence of statements, execute the first statement. If that statement does not redirect control, then proceed to execute the rest of the sequence of statements, if any remain.

This definition exposes the essential structure of a recursively defined *sequence*: a sequence can be decomposed into its first element and the rest of its elements. The "rest" of a sequence of statements is itself a sequence of statements! Thus, we can recursively apply this execution rule. This view of sequences as recursive data structures will appear again in later chapters.

The important consequence of this rule is that statements are executed in order, but later statements may never be reached, because of redirected control.

Practical Guidance. When indenting a suite, all lines must be indented the same amount and in the same way (spaces, not tabs). Any variation in indentation will cause an error.

1.5.3 Defining Functions II: Local Assignment

Originally, we stated that the body of a user-defined function consisted only of a `return` statement with a single return expression. In fact, functions can define a sequence of operations that extends beyond a single expression. The structure of compound Python statements naturally allows us to extend our concept of a function body to multiple statements.

Whenever a user-defined function is applied, the sequence of clauses in the suite of its definition is executed in a local environment. A `return` statement redirects control: the process of function application terminates whenever the first `return` statement is executed, and the value of the `return` expression is the returned value of the function being applied.

Thus, assignment statements can now appear within a function body. For instance, this function returns the absolute difference between two quantities as a percentage of the first, using a two-step calculation:

```
>>> def percent_difference(x, y):
        difference = abs(x-y)
        return 100 * difference / x
>>> percent_difference(40, 50)
25.0
```

The effect of an assignment statement is to bind a name to a value in the *first* frame of the current environment. As a consequence, assignment statements within a function body cannot affect the global frame. The fact that functions can only manipulate their local environment is critical to creating *modular* programs, in which pure functions interact only via the values they take and return.

Of course, the `percent_difference` function could be written as a single expression, as shown below, but the return expression is more complex.

```
>>> def percent_difference(x, y):
        return 100 * abs(x-y) / x
```

So far, local assignment hasn't increased the expressive power of our function definitions. It will do so, when combined with the control statements below. In addition, local assignment also plays a critical role in clarifying the meaning of complex expressions by assigning names to intermediate quantities.

New environment Feature: Local assignment.

1.5.4 Conditional Statements

Python has a built-in function for computing absolute values.

```
>>> abs(-2)
2
```

We would like to be able to implement such a function ourselves, but we cannot currently define a function that has a test and a choice. We would like to express that if `x` is positive, `abs(x)` returns `x`. Furthermore, if `x` is 0, `abs(x)` returns 0. Otherwise, `abs(x)` returns `-x`. In Python, we can express this choice with a conditional statement.

```
>>> def absolute_value(x):
        """Compute abs(x)."""
        if x > 0:
            return x
        elif x == 0:
            return 0
        else:
            return -x
```

```
>>> absolute_value(-2) == abs(-2)
True
```

This implementation of `absolute_value` raises several important issues.

Conditional statements. A conditional statement in Python consist of a series of headers and suites: a required `if` clause, an optional sequence of `elif` clauses, and finally an optional `else` clause:

```
if <expression>:
    <suite>
elif <expression>:
    <suite>
else:
    <suite>
```

When executing a conditional statement, each clause is considered in order.

1. Evaluate the header's expression.
2. If it is a true value, execute the suite. Then, skip over all subsequent clauses in the conditional statement.

If the `else` clause is reached (which only happens if all `if` and `elif` expressions evaluate to false values), its suite is executed.

Boolean contexts. Above, the execution procedures mention "a false value" and "a true value." The expressions inside the header statements of conditional blocks are said to be in *boolean contexts*: their truth values matter to control flow, but otherwise their values can never be assigned or returned. Python includes several false values, including 0, `None`, and the *boolean* value `False`. All other numbers are true values. In Chapter 2, we will see that every native data type in Python has both true and false values.

Boolean values. Python has two boolean values, called `True` and `False`. Boolean values represent truth values in logical expressions. The built-in comparison operations, >, <, >=, <=, ==, != , return these values.

```
>>> 4 < 2
False
>>> 5 >= 5
True
```

This second example reads "5 is greater than or equal to 5", and corresponds to the function `ge` in the `operator` module.

```
>>> 0 == -0
True
```

This final example reads "0 equals -0", and corresponds to `eq` in the `operator` module. Notice that Python distinguishes assignment (`=`) from equality testing (`==`), a convention shared across many programming languages.

Boolean operators. Three basic logical operators are also built into Python:

```
>>> True and False
False
>>> True or False
True
>>> not False
True
```

Logical expressions have corresponding evaluation procedures. These procedures exploit the fact that the truth value of a logical expression can sometimes be determined without evaluating all of its subexpressions, a feature called *short-circuiting*.

To evaluate the expression `<left> and <right>` :

1. Evaluate the subexpression `<left>` .
2. If the result is a false value `v` , then the expression evaluates to `v` .
3. Otherwise, the expression evaluates to the value of the subexpression `<right>` .

To evaluate the expression `<left> or <right>` :

1. Evaluate the subexpression `<left>` .
2. If the result is a true value `v` , then the expression evaluates to `v` .
3. Otherwise, the expression evaluates to the value of the subexpression `<right>` .

To evaluate the expression `not <exp>` :

1. Evaluate `<exp>` ; The value is `True` if the result is a false value, and `False` otherwise.

These values, rules, and operators provide us with a way to combine the results of tests. Functions that perform tests and return boolean values typically begin with `is` , not followed by an underscore (e.g., `isfinite` , `isdigit` , `isinstance` , etc.).

1.5.5 Iteration

In addition to selecting which statements to execute, control statements are used to express repetition. If each line of code we wrote were only executed once, programming would be a very unproductive exercise. Only through repeated execution of statements do we unlock the potential of computers to make us powerful. We have already seen one form of repetition: a function can be applied many times, although it is only defined once. Iterative control structures are another mechanism for executing the same statements many times.

Consider the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

Each value is constructed by repeatedly applying the sum-previous-two rule. To build up the nth value, we need to track how many values we've created (`k`), along with the kth value (`curr`) and its predecessor (`pred`), like so:

```
>>> def fib(n):
        """Compute the nth Fibonacci number, for n >= 2."""
        pred, curr = 0, 1   # Fibonacci numbers
        k = 2               # Position of curr in the sequence
        while k < n:
            pred, curr = curr, pred + curr  # Re-bind pred and curr
            k = k + 1                       # Re-bind k
        return curr
```

```
>>> fib(8)
13
```

Remember that commas seperate multiple names and values in an assignment statement. The line:

```
pred, curr = curr, pred + curr
```

has the effect of rebinding the name `pred` to the value of `curr`, and simultanously rebinding `curr` to the value of `pred + curr`. All of the expressions to the right of `=` are evaluated before any rebinding takes place.

A `while` clause contains a header expression followed by a suite:

```
while <expression>:
    <suite>
```

To execute a `while` clause:

1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then return to step 1.

In step 2, the entire suite of the `while` clause is executed before the header expression is evaluated again.

In order to prevent the suite of a `while` clause from being executed indefinitely, the suite should always change the state of the environment in each pass.

A `while` statement that does not terminate is called an infinite loop. Press `<Control>-C` to force Python to stop looping.

1.5.6 Practical Guidance: Testing

Testing a function is the act of verifying that the function's behavior matches expectations. Our language of functions is now sufficiently complex that we need to start testing our implementations.

A *test* is a mechanism for systematically performing this verification. Tests typically take the form of another function that contains one or more sample calls to the function being tested. The returned value is then verified against an expected result. Unlike most functions, which are meant to be general, tests involve selecting and validating calls with specific argument values. Tests also serve as documentation: they demonstrate how to call a function, and what argument values are appropriate.

Note that we have also used the word "test" as a technical term for the expression in the header of an `if` or `while` statement. It should be clear from context when we use the word "test" to denote an expression, and when we use it to denote a verification mechanism.

Assertions. Programmers use `assert` statements to verify expectations, such as the output of a function being tested. An `assert` statement has an expression in a boolean context, followed by a quoted line of text (single or double quotes are both fine, but be consistent) that will be displayed if the expression evaluates to a false value.

```
>>> assert fib(8) == 13, 'The 8th Fibonacci number should be 13'
```

When the expression being asserted evaluates to a true value, executing an assert statement has no effect. When it is a false value, `assert` causes an error that halts execution.

A test function for `fib` should test several arguments, including extreme values of `n`.

```
>>> def fib_test():
        assert fib(2) == 1, 'The 2nd Fibonacci number should be 1'
        assert fib(3) == 1, 'The 3nd Fibonacci number should be 1'
        assert fib(50) == 7778742049, 'Error at the 50th Fibonacci number'
```

When writing Python in files, rather than directly into the interpreter, tests should be written in the same file or a neighboring file with the suffix `_test.py`.

Doctests. Python provides a convenient method for placing simple tests directly in the docstring of a function. The first line of a docstring should contain a one-line description of the function, followed by a blank line. A detailed description of arguments and behavior may follow. In addition, the docstring may include a sample interactive session that calls the function:

```
>>> def sum_naturals(n):
        """Return the sum of the first n natural numbers

        >>> sum_naturals(10)
        55
        >>> sum_naturals(100)
        5050
        """
        total, k = 0, 1
        while k <= n:
          total, k = total + k, k + 1
        return total
```

Then, the interaction can be verified via the doctest module. Below, the `globals` function returns a representation of the global environment, which the interpreter needs in order to evaluate expressions.

```
>>> from doctest import run_docstring_examples
>>> run_docstring_examples(sum_naturals, globals())
```

When writing Python in files, all doctests in a file can be run by starting Python with the doctest command line option:

```
python3 -m doctest <python_source_file>
```

The key to effective testing is to write (and run) tests immediately after (or even before) implementing new functions. A test that applies a single function is called a *unit test*. Exhaustive unit testing is a hallmark of good program design.

1.6 Higher-Order Functions

We have seen that functions are, in effect, abstractions that describe compound operations independent of the particular values of their arguments. In `square`,

```
>>> def square(x):
        return x * x
```

we are not talking about the square of a particular number, but rather about a method for obtaining the square of any number. Of course we could get along without ever defining this function, by always writing expressions such as

```
>>> 3 * 3
9
>>> 5 * 5
25
```

and never mentioning `square` explicitly. This practice would suffice for simple computations like `square`, but would become arduous for more complex examples. In general, lacking function definition would put us at the disadvantage of forcing us to work always at the level of the particular operations that happen to be primitives in the language (multiplication, in this case) rather than in terms of higher-level operations. Our programs would be able to compute squares, but our language would lack the ability to express the concept of squaring. One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. Functions provide this ability.

As we will see in the following examples, there are common programming patterns that recur in code, but are used with a number of different functions. These patterns can also be abstracted, by giving them names.

To express certain general patterns as named concepts, we will need to construct functions that can accept other functions as arguments or return functions as values. Functions that manipulate functions are called higher-order functions. This section shows how higher-order functions can serve as powerful abstraction mechanisms, vastly increasing the expressive power of our language.

1.6.1 Functions as Arguments

Consider the following three functions, which all compute summations. The first, `sum_naturals`, computes the sum of natural numbers up to `n`:

```
>>> def sum_naturals(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k, k + 1
        return total
```

```
>>> sum_naturals(100)
5050
```

The second, `sum_cubes`, computes the sum of the cubes of natural numbers up to `n`.

```
>>> def sum_cubes(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + pow(k, 3), k + 1
        return total
```

```
>>> sum_cubes(100)
25502500
```

The third, `pi_sum`, computes the sum of terms in the series

$$\frac{8}{1 \cdot 3} + \frac{8}{5 \cdot 7} + \frac{8}{9 \cdot 11} + \dots$$

which converges to pi very slowly.

```
>>> def pi_sum(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + 8 / (k * (k + 2)), k + 4
        return total
```

```
>>> pi_sum(100)
3.121594652591009
```

These three functions clearly share a common underlying pattern. They are for the most part identical, differing only in name, the function of `k` used to compute the term to be added, and the function that provides the next value of `k`. We could generate each of the functions by filling in slots in the same template:

```
def <name>(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + <term>(k), <next>(k)
    return total
```

The presence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Each of these functions is a summation of terms. As program designers, we would like our language to be powerful enough so that we can write a function that expresses the concept of summation itself rather than only functions that compute particular sums. We can do so readily in Python by taking the common template shown above and transforming the "slots" into formal parameters:

```
>>> def summation(n, term, next):
        total, k = 0, 1
        while k <= n:
            total, k = total + term(k), next(k)
        return total
```

Notice that `summation` takes as its arguments the upper bound `n` together with the functions `term` and `next`. We can use `summation` just as we would any function, and it expresses summations succinctly:

```
>>> def cube(k):
        return pow(k, 3)
```

```
>>> def successor(k):
        return k + 1
```

```
>>> def sum_cubes(n):
        return summation(n, cube, successor)
```

```
>>> sum_cubes(3)
36
```

Using an `identity` function that returns its argument, we can also sum integers.

```
>>> def identity(k):
        return k
```

```
>>> def sum_naturals(n):
        return summation(n, identity, successor)
```

```
>>> sum_naturals(10)
55
```

We can also define `pi_sum` piece by piece, using our `summation` abstraction to combine components.

```
>>> def pi_term(k):
        denominator = k * (k + 2)
        return 8 / denominator
```

```
>>> def pi_next(k):
        return k + 4
```

```
>>> def pi_sum(n):
        return summation(n, pi_term, pi_next)
```

```
>>> pi_sum(1e6)
3.1415906535898936
```

1.6.2 Functions as General Methods

We introduced user-defined functions as a mechanism for abstracting patterns of numerical operations so as to make them independent of the particular numbers involved. With higher-order functions, we begin to see a more powerful kind of abstraction: some functions express general methods of computation, independent of the particular functions they call.

Despite this conceptual extension of what a function means, our environment model of how to evaluate a call expression extends gracefully to the case of higher-order functions, without change. When a user-defined function is applied to some arguments, the formal parameters are bound to the values of those arguments (which may be functions) in a new local frame.

Consider the following example, which implements a general method for iterative improvement and uses it to compute the golden ratio. An iterative improvement algorithm begins with a `guess` of a solution to an equation. It repeatedly applies an `update` function to improve that guess, and applies a `test` to check whether the current `guess` is "close enough" to be considered correct.

```
>>> def iter_improve(update, test, guess=1):
        while not test(guess):
            guess = update(guess)
        return guess
```

The `test` function typically checks whether two functions, `f` and `g`, are near to each other for the value `guess`. Testing whether `f(x)` is near to `g(x)` is again a general method of computation.

```
>>> def near(x, f, g):
        return approx_eq(f(x), g(x))
```

A common way to test for approximate equality in programs is to compare the absolute value of the difference between numbers to a small tolerance value.

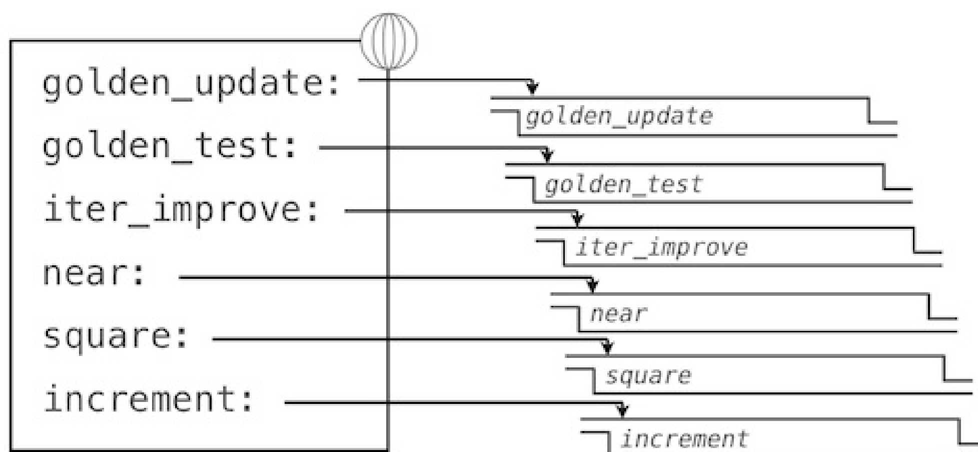
```
>>> def approx_eq(x, y, tolerance=1e-5):
        return abs(x - y) < tolerance
```

The golden ratio, often called phi, is a number that appears frequently in nature, art, and architecture. It can be computed via `iter_improve` using the `golden_update` , and it converges when its successor is equal to its square.

```
>>> def golden_update(guess):
        return 1/guess + 1
```

```
>>> def golden_test(guess):
        return near(guess, square, successor)
```

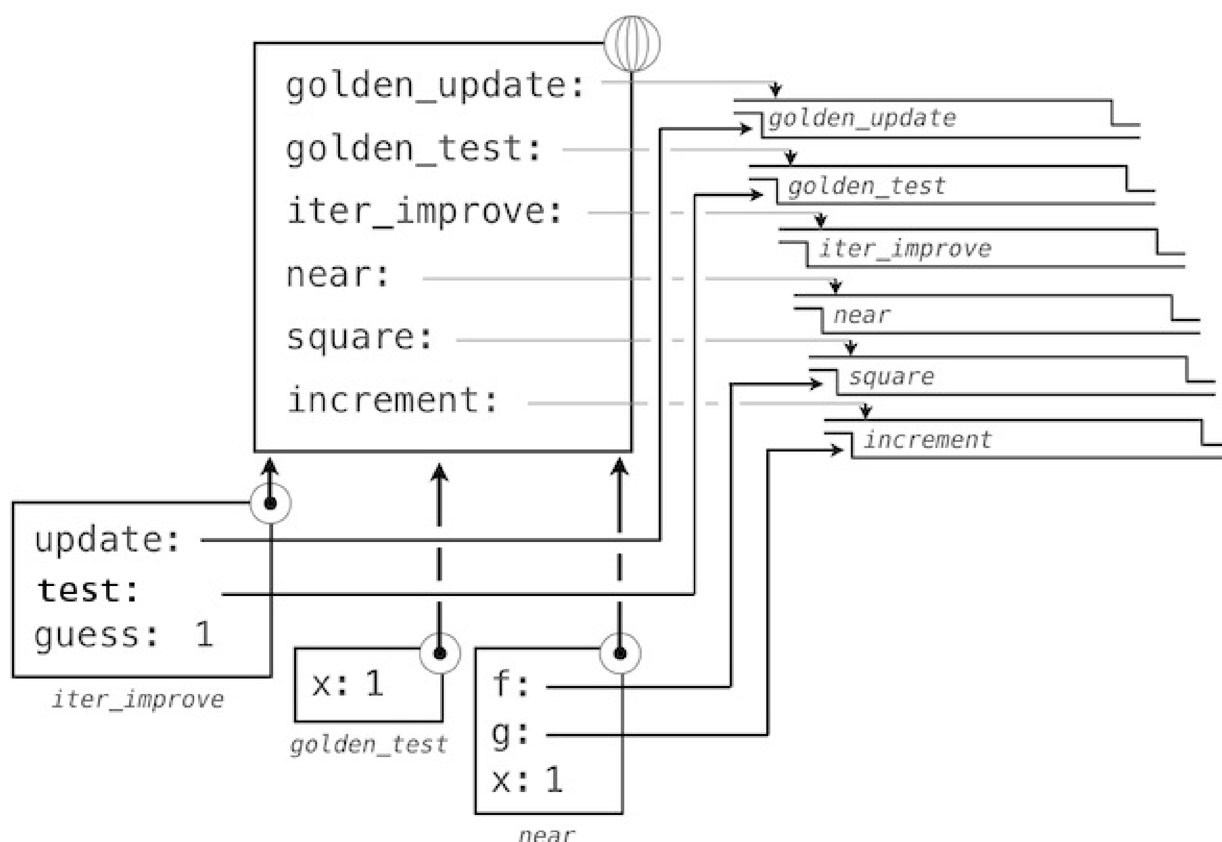
At this point, we have added several bindings to the global frame. The depictions of function values are abbreviated for clarity.



Calling `iter_improve` with the arguments `golden_update` and `golden_test` will compute an approximation to the golden ratio.

```
>>> iter_improve(golden_update, golden_test)
1.6180371352785146
```

By tracing through the steps of our evaluation procedure, we can see how this result is computed. First, a local frame for `iter_improve` is constructed with bindings for `update` , `test` , and `guess` . In the body of `iter_improve` , the name `test` is bound to `golden_test` , which is called on the initial value of `guess` . In turn, `golden_test` calls `near` , creating a third local frame that binds the formal parameters `f` and `g` to `square` and `successor` .



Completing the evaluation of `near`, we see that the `golden_test` is `False` because 1 is not close to 2. Hence, evaluation proceeds with the suite of the `while` clause, and this mechanical process repeats several times.

This extended example illustrates two related big ideas in computer science. First, naming and functions allow us to abstract away a vast amount of complexity. While each function definition has been trivial, the computational process set in motion by our evaluation procedure appears quite intricate, and we didn't even illustrate the whole thing. Second, it is only by virtue of the fact that we have an extremely general evaluation procedure that small components can be composed into complex processes. Understanding that procedure allows us to validate and inspect the process we have created.

As always, our new general method `iter_improve` needs a test to check its correctness. The golden ratio can provide such a test, because it also has an exact closed-form solution, which we can compare to this iterative result.

```
>>> phi = 1/2 + pow(5, 1/2)/2
>>> def near_test():
        assert near(phi, square, successor), 'phi * phi is not near phi + 1'
```

```
>>> def iter_improve_test():
        approx_phi = iter_improve(golden_update, golden_test)
        assert approx_eq(phi, approx_phi), 'phi differs from its approximation'
```

New environment Feature: Higher-order functions.

Extra for experts. We left out a step in the justification of our test. For what range of tolerance values `e` can you prove that if `near(x, square, successor)` is true with `tolerance` value `e`, then `approx_eq(phi, x)` is true with the same tolerance?

1.6.3 Defining Functions III: Nested Definitions

The above examples demonstrate how the ability to pass functions as arguments significantly enhances the expressive power of our programming language. Each general concept or equation maps onto its own short function. One negative consequence of this approach to programming is that the global frame becomes cluttered with names of small functions. Another problem is that we are constrained by particular function signatures: the `update` argument to `iter_improve` must take exactly one argument. In Python, nested function definitions address both of these problems, but require us to amend our environment model slightly.

Let's consider a new problem: computing the square root of a number. Repeated application of the following update converges to the square root of `x`:

```
>>> def average(x, y):
        return (x + y)/2
```

```
>>> def sqrt_update(guess, x):
        return average(guess, x/guess)
```

This two-argument update function is incompatible with `iter_improve`, and it just provides an intermediate value; we really only care about taking square roots in the end. The solution to both of these issues is to place function definitions inside the body of other definitions.

```
>>> def square_root(x):
        def update(guess):
            return average(guess, x/guess)
        def test(guess):
            return approx_eq(square(guess), x)
        return iter_improve(update, test)
```

Like local assignment, local `def` statements only affect the current local frame. These functions are only in scope while `square_root` is being evaluated. Consistent with our evaluation procedure, these local `def` statements don't even get evaluated until `square_root` is called.

Lexical scope. Locally defined functions also have access to the name bindings in the scope in which they are defined. In this example, `update` refers to the name `x`, which is a formal parameter of its enclosing function `square_root`. This discipline of sharing names

among nested definitions is called *lexical scoping*. Critically, the inner functions have access to the names in the environment where they are defined (not where they are called).

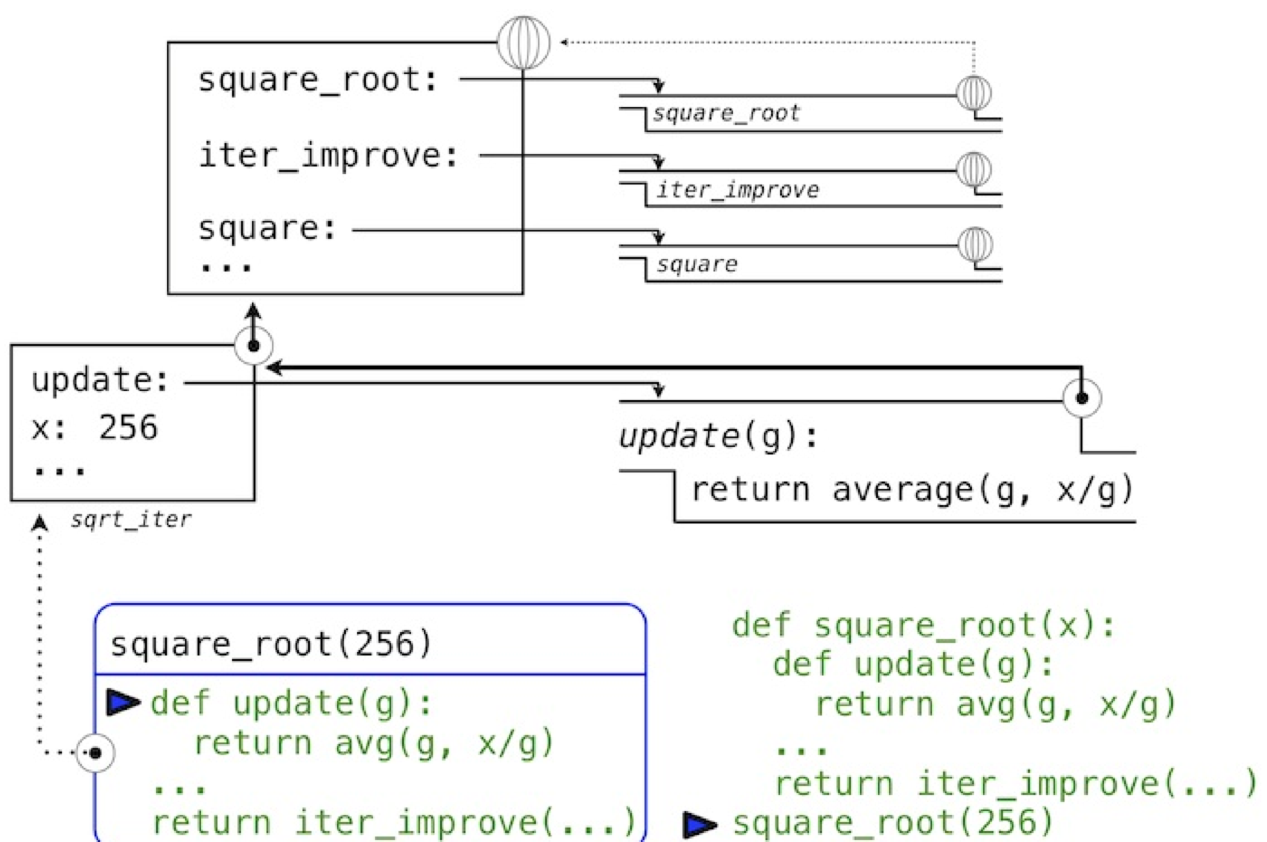
We require two extensions to our environment model to enable lexical scoping.

1. Each user-defined function has an associated environment: the environment in which it was defined.
2. When a user-defined function is called, its local frame extends the environment associated with the function.

Previous to `square_root`, all functions were defined in the global environment, and so they were all associated with the global environment. When we evaluate the first two clauses of `square_root`, we create functions that are associated with a local environment. In the call

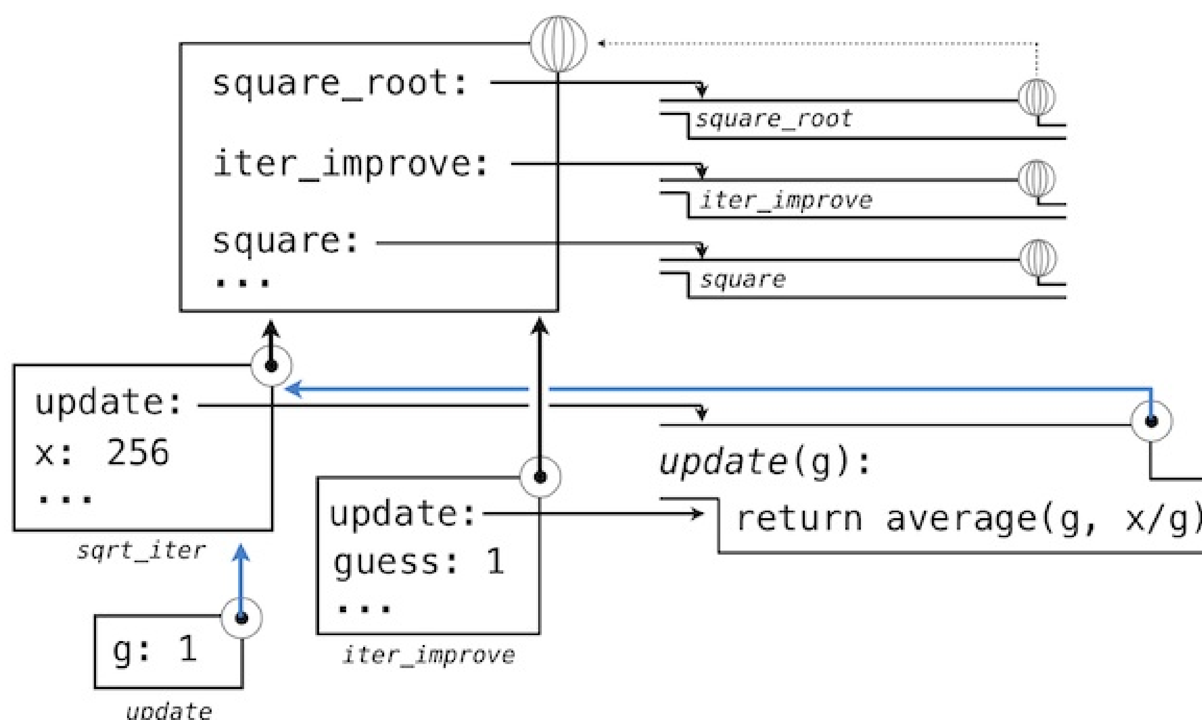
```
>>> square_root(256)
16.00000000000039
```

the environment first adds a local frame for `square_root` and evaluates the `def` statements for `update` and `test` (only `update` is shown).



Subsequently, the name `update` resolves to this newly defined function, which is passed as an argument to `iter_improve`. Within the body of `iter_improve`, we must apply our `update` function to the initial `guess` of 1. This final application creates an environment for `update`

that begins with a local frame containing only `g`, but with the preceding frame for `square_root` still containing a binding for `x`.



The most crucial part of this evaluation procedure is the transfer of an environment associated with a function to the local frame in which that function is evaluated. This transfer is highlighted by the blue arrows in this diagram.

In this way, the body of `update` can resolve a value for `x`. Hence, we realize two key advantages of lexical scoping in Python.

- The names of a local function do not interfere with names external to the function in which it is defined, because the local function name will be bound in the current local environment in which it is defined, rather than the global environment.
- A local function can access the environment of the enclosing function. This is because the body of the local function is evaluated in an environment that extends the evaluation environment in which it is defined.

The `update` function carries with it some data: the values referenced in the environment in which it was defined. Because they enclose information in this way, locally defined functions are often called *closures*.

New environment Feature: Local function definition.

1.6.4 Functions as Returned Values

We can achieve even more expressive power in our programs by creating functions whose returned values are themselves functions. An important feature of lexically scoped programming languages is that locally defined functions keep their associated environment when they are returned. The following example illustrates the utility of this feature.

With many simple functions defined, function *composition* is a natural method of combination to include in our programming language. That is, given two functions `f(x)` and `g(x)`, we might want to define `h(x) = f(g(x))`. We can define function composition using our existing tools:

```
>>> def compose1(f, g):
        def h(x):
            return f(g(x))
        return h
```

```
>>> add_one_and_square = compose1(square, successor)
>>> add_one_and_square(12)
169
```

The `1` in `compose1` indicates that the composed functions and returned result all take `1` argument. This naming convention isn't enforced by the interpreter; the `1` is just part of the function name.

At this point, we begin to observe the benefits of our investment in a rich model of computation. No modifications to our environment model are required to support our ability to return functions in this way.

1.6.5 Lambda Expressions

So far, every time we want to define a new function, we need to give it a name. But for other types of expressions, we don’t need to associate intermediate products with a name. That is, we can compute `a*b + c*d` without having to name the subexpressions `a*b` or `c*d`, or the full expression. In Python, we can create function values on the fly using `lambda` expressions, which evaluate to unnamed functions. A lambda expression evaluates to a function that has a single return expression as its body. Assignment and control statements are not allowed.

Lambda expressions are limited: They are only useful for simple, one-line functions that evaluate and return a single expression. In those special cases where they apply, lambda expressions can be quite expressive.

```
>>> def compose1(f,g):
        return lambda x: f(g(x))
```

We can understand the structure of a `lambda` expression by constructing a corresponding English sentence:

```
     lambda            x            :          f(g(x))
"A function that    takes x    and returns     f(g(x))"
```

Some programmers find that using unnamed functions from lambda expressions is shorter and more direct. However, compound `lambda` expressions are notoriously illegible, despite their brevity. The following definition is correct, but some programmers have trouble understanding it quickly.

```
>>> compose1 = lambda f,g: lambda x: f(g(x))
```

In general, Python style prefers explicit `def` statements to lambda expressions, but allows them in cases where a simple function is needed as an argument or return value.

Such stylistic rules are merely guidelines; you can program any way you wish. However, as you write programs, think about the audience of people who might read your program one day. If you can make your program easier to interpret, you will do those people a favor.

The term *lambda* is a historical accident resulting from the incompatibility of written mathematical notation and the constraints of early type-setting systems.

> It may seem perverse to use lambda to introduce a procedure/function. The notation goes back to Alonzo Church, who in the 1930's started with a "hat" symbol; he wrote the square function as "? . y × y". But frustrated typographers moved the hat to the left of the parameter and changed it to a capital lambda: "Λy . y × y"; from there the capital lambda was changed to lowercase, and now we see "λy . y × y" in math books and `(lambda (y) (* y y))` in Lisp. > > —Peter Norvig (norvig.com/lispy2.html)

Despite their unusual etymology, `lambda` expressions and the corresponding formal language for function application, the *lambda calculus*, are fundamental computer science concepts shared far beyond the Python programming community. We will revisit this topic when we study the design of interpreters in Chapter 3.

1.6.6 Example: Newton's Method

This final extended example shows how function values, local defintions, and lambda expressions can work together to express general ideas concisely.

Newton's method is a classic iterative approach to finding the arguments of a mathematical function that yield a return value of 0. These values are called *roots* of a single-argument mathematical function. Finding a root of a function is often equivalent to solving a related

math problem.

- The square root of 16 is the value `x` such that: `square(x) - 16 = 0`
- The log base 2 of 32 (i.e., the exponent to which we would raise 2 to get 32) is the value `x` such that: `pow(2, x) - 32 = 0`

Thus, a general method for finding roots will also provide us an algorithm to compute square roots and logarithms. Moreover, the equations for which we want to compute roots only contain simpler operations: multiplication and exponentiation.

A comment before we proceed: it is easy to take for granted the fact that we know how to compute square roots and logarithms. Not just Python, but your phone, your pocket calculator, and perhaps even your watch can do so for you. However, part of learning computer science is understanding how quantities like these can be computed, and the general approach presented here is applicable to solving a large class of equations beyond those built into Python.

Before even beginning to understand Newton's method, we can start programming; this is the power of functional abstractions. We simply translate our previous statements into code.

```
>>> def square_root(a):
        return find_root(lambda x: square(x) - a)
```

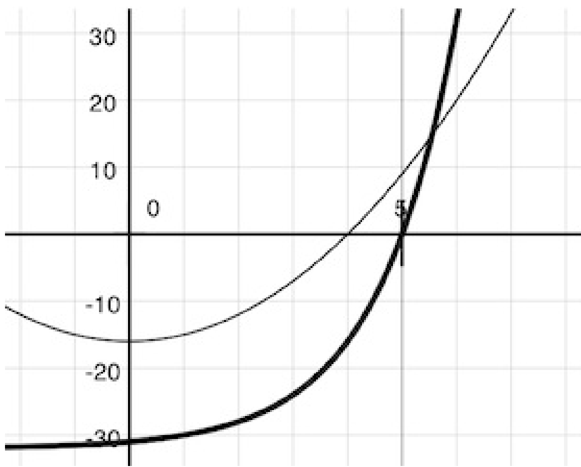
```
>>> def logarithm(a, base=2):
        return find_root(lambda x: pow(base, x) - a)
```

Of course, we cannot apply any of these functions until we define *find_root*, and so we need to understand how Newton's method works.

Newton's method is also an iterative improvement algorithm: it improves a guess of the root for any function that is *differentiable*. Notice that both of our functions of interest change smoothly; graphing `x` versus `f(x)` for

- `f(x) = square(x) - 16` (light curve)
- `f(x) = pow(2, x) - 32` (dark curve)

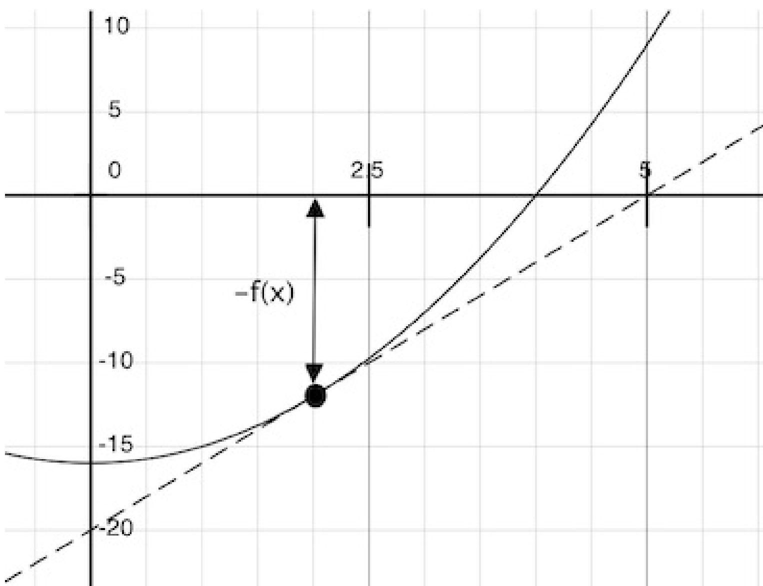
on a 2-dimensional plane shows that both functions produce a smooth curve without kinks that crosses `0` at the appropriate point.



Because they are smooth (differentiable), these curves can be approximated by a line at any point. Newton's method follows these linear approximations to find function roots.

Imagine a line through the point `(x, f(x))` that has the same slope as the curve for function `f(x)` at that point. Such a line is called the *tangent*, and its slope is called the *derivative* of `f` at `x`.

This line's slope is the ratio of the change in function value to the change in function argument. Hence, translating `x` by `f(x)` divided by the slope will give the argument value at which this tangent line touches 0.



Our Newton update expresses the computational process of following this tangent line to 0. We approximate the derivative of the function by computing its slope over a very small interval.

```
>>> def approx_derivative(f, x, delta=1e-5):
        df = f(x + delta) - f(x)
        return df/delta
```

```
>>> def newton_update(f):
        def update(x):
            return x - f(x) / approx_derivative(f, x)
        return update
```

Finally, we can define the `find_root` function in terms of `newton_update`, our iterative improvement algorithm, and a test to see if `f(x)` is near 0. We supply a larger initial guess to improve performance for `logarithm`.

```
>>> def find_root(f, initial_guess=10):
        def test(x):
            return approx_eq(f(x), 0)
        return iter_improve(newton_update(f), test, initial_guess)
```

```
>>> square_root(16)
4.000000000026422
>>> logarithm(32, 2)
5.000000094858201
```

As you experiment with Newton's method, be aware that it will not always converge. The initial guess of `iter_improve` must be sufficiently close to the root, and various conditions about the function must be met. Despite this shortcoming, Newton's method is a powerful general computational method for solving differentiable equations. In fact, very fast algorithms for logarithms and large integer division employ variants of the technique.

1.6.7 Abstractions and First-Class Functions

We began this section with the observation that user-defined functions are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit elements in our programming language. Now we've seen how higher-order functions permit us to manipulate these general methods to create further abstractions.

As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs, to build upon them, and generalize them to create more powerful abstractions. This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts. The significance of higher-order functions is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have first-class status. Some of the "rights and privileges" of first-class elements are:

1. They may be bound to names.
2. They may be passed as arguments to functions.
3. They may be returned as the results of functions.
4. They may be included in data structures.

Python awards functions full first-class status, and the resulting gain in expressive power is enormous. Control structures, on the other hand, do not: you cannot pass `if` to a function the way you can `sum`.

1.6.8 Function Decorators

Python provides special syntax to apply higher-order functions as part of executing a `def` statement, called a decorator. Perhaps the most common example is a trace.

```
>>> def trace1(fn):
        def wrapped(x):
            print('-> ', fn, '(', x, ')')
            return fn(x)
        return wrapped
```

```
>>> @trace1
    def triple(x):
        return 3 * x
```

```
>>> triple(12)
->  <function triple at 0x102a39848> ( 12 )
36
```

In this example, A higher-order function `trace1` is defined, which returns a function that precedes a call to its argument with a `print` statement that outputs the argument. The `def` statement for `triple` has an annototation, `@trace1`, which affects the execution rule for `def`. As usual, the function `triple` is created. However, the name `triple` is not bound to this function. Instead, the name `triple` is bound to the returned function value of calling `trace1` on the newly defined `triple` function. In code, this decorator is equivalent to:

```
>>> def triple(x):
        return 3 * x
```

```
>>> triple = trace1(triple)
```

In the projects for this course, decorators are used for tracing, as well as selecting which functions to call when a program is run from the command line.

Extra for experts. The actual rule is that the decorator symbol `@` may be followed by an expression (`@trace1` is just a simple expression consisting of a single name). Any expression producing a suitable value is allowed. For example, with a suitable definition, you could define a decorator `check_range` so that decorating a function definition with `@check_range(1, 10)` would cause the function's results to be checked to make sure they are integers between 1 and 10. The call `check_range(1,10)` would return a function that would then be applied to the newly defined function before it is bound to the name in the `def` statement. A short tutorial on decorators by Ariel Ortiz gives further examples for interested students.

Chapter 2: Building Abstractions with Objects

Contents

2.1 Introduction

We concentrated in Chapter 1 on computational processes and on the role of functions in program design. We saw how to use primitive data (numbers) and primitive operations (arithmetic operations), how to form compound functions through composition and control, and how to create functional abstractions by giving names to processes. We also saw that higher-order functions enhance the power of our language by enabling us to manipulate, and thereby to reason, in terms of general methods of computation. This is much of the essence of programming.

This chapter focuses on data. Data allow us to represent and manipulate information about the world using the computational tools we have acquired so far. Programs without data structures may suffice for exploring mathematical properties. But real-world phenomena, such as documents, relationships, cities, and weather patterns, all have complex structure that is best represented using *compound data types*. With structured data, programs can simulate and reason about virtually any domain of human knowledge and experience. Thanks to the explosive growth of the Internet, a vast amount of structured information about the world is freely available to us all online.

2.1.1 The Object Metaphor

In the beginning of this course, we distinguished between functions and data: functions performed operations and data were operated upon. When we included function values among our data, we acknowledged that data too can have behavior. Functions could be operated upon like data, but could also be called to perform computation.

In this course, *objects* will serve as our central programming metaphor for data values that also have behavior. Objects represent information, but also *behave* like the abstract concepts that they represent. The logic of how an object interacts with other objects is bundled along with the information that encodes the object's value. When an object is printed, it knows how to spell itself out in letters and numbers. If an object is composed of parts, it knows how to reveal those parts on demand. Objects are both information and processes, bundled together to represent the properties, interactions, and behaviors of complex things.

The object metaphor is implemented in Python through specialized object syntax and associated terminology, which we can introduce by example. A date is a kind of simple object.

```
>>> from datetime import date
```

The name `date` is bound to a *class*. A class represents a kind of object. Individual dates are called *instances* of that class, and they can be *constructed* by calling the class as a function on arguments that characterize the instance.

```
>>> today = date(2011, 9, 12)
```

While `today` was constructed from primitive numbers, it behaves like a date. For instance, subtracting it from another date will give a time difference, which we can display as a line of text by calling `str`.

```
>>> str(date(2011, 12, 2) - today)
'81 days, 0:00:00'
```

Objects have *attributes*, which are named values that are part of the object. In Python, we use dot notation to designated an attribute of an object.

> <expression> . <name>

Above, the `<expression>` evaluates to an object, and `<name>` is the name of an attribute for that object.

Unlike the names that we have considered so far, these attribute names are not available in the general environment. Instead, attribute names are particular to the object instance preceding the dot.

```
>>> today.year
2011
```

Objects also have *methods*, which are function-valued attributes. Metaphorically, the object "knows" how to carry out those methods. Methods compute their results from both their arguments and their object. For example, The `strftime` method of `today` takes a single argument that specifies how to display a date (e.g., `%A` means that the day of the week should be spelled out in full).

```
>>> today.strftime('%A, %B %d')
'Monday, September 12'
```

Computing the return value of `strftime` requires two inputs: the string that describes the format of the output and the date information bundled into `today`. Date-specific logic is applied within this method to yield this result. We never stated that the 12th of September,

2011, was a Monday, but knowing one's weekday is part of what it means to be a date. By bundling behavior and information together, this Python object offers us a convincing, self-contained abstraction of a date.

Dot notation provides another form of combined expression in Python. Dot notation also has a well-defined evaluation procedure. However, developing a precise account of how dot notation is evaluated will have to wait until we introduce the full paradigm of object-oriented programming over the next several sections.

Even though we haven't described precisely how objects work yet, it is time to start thinking about data as objects now, because in Python every value is an object.

2.1.2 Native Data Types

Every object in Python has a *type*. The `type` function allows us to inspect the type of an object.

```
>>> type(today)
<class 'datetime.date'>
```

So far, the only kinds of objects we have studied are numbers, functions, Booleans, and now dates. We also briefly encountered sets and strings, but we will need to study those in more depth. There are many other kinds of objects --- sounds, images, locations, data connections, etc. --- most of which can be defined by the means of combination and abstraction that we develop in this chapter. Python has only a handful of primitive or *native* data types built into the language.

Native data types have the following properties:

1. There are primitive expressions that evaluate to objects of these types, called *literals*.
2. There are built-in functions, operators, and methods to manipulate these objects.

As we have seen, numbers are native; numeric literals evaluate to numbers, and mathematical operators manipulate number objects.

```
>>> 12 + 3000000000000000000000000
3000000000000000000000012
```

In fact, Python includes three native numeric types: integers (`int`), real numbers (`float`), and complex numbers (`complex`).

```
>>> type(2)
<class 'int'>
>>> type(1.5)
<class 'float'>
>>> type(1+1j)
<class 'complex'>
```

The name `float` comes from the way in which real numbers are represented in Python: a "floating point" representation. While the details of how numbers are represented is not a topic for this course, some high-level differences between `int` and `float` objects are important to know. In particular, `int` objects can only represent integers, but they represent them exactly, without any approximation. On the other hand, `float` objects can represent a wide range of fractional numbers, but not all rational numbers are representable. Nonetheless, float objects are often used to represent real and rational numbers approximately, up to some number of significant figures.

Further reading. The following sections introduce more of Python's native data types, focusing on the role they play in creating useful data abstractions. A chapter on native data types in Dive Into Python 3 gives a pragmatic overview of all Python's native data types and how to use them effectively, including numerous usage examples and practical tips. You needn't read that chapter now, but consider it a valuable reference.

2.2 Data Abstraction

As we consider the wide set of things in the world that we would like to represent in our programs, we find that most of them have compound structure. A date has a year, a month, and a day; a geographic position has a latitude and a longitude. To represent positions, we would like our programming language to have the capacity to "glue together" a latitude and longitude to form a pair --- a *compound data* value --- that our programs could manipulate in a way that would be consistent with the fact that we regard a position as a single conceptual unit, which has two parts.

The use of compound data also enables us to increase the modularity of our programs. If we can manipulate geographic positions directly as objects in their own right, then we can separate the part of our program that deals with values per se from the details of how those values may be represented. The general technique of isolating the parts of a program that deal with how data are represented from the parts of a program that deal with how those data are manipulated is a powerful design methodology called *data abstraction*. Data abstraction makes programs much easier to design, maintain, and modify.

Data abstraction is similar in character to functional abstraction. When we create a functional abstraction, the details of how a function is implemented can be suppressed, and the particular function itself can be replaced by any other function with the same overall behavior. In other words, we can make an abstraction that separates the way the function is used from the details of how the function is implemented. Analogously, data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed.

The basic idea of data abstraction is to structure programs so that they operate on abstract data. That is, our programs should use data in such a way as to make as few assumptions about the data as possible. At the same time, a concrete data representation is defined, independently of the programs that use the data. The interface between these two parts of our system will be a set of functions, called selectors and constructors, that implement the abstract data in terms of the concrete representation. To illustrate this technique, we will consider how to design a set of functions for manipulating rational numbers.

As you read the next few sections, keep in mind that most Python code written today uses very high-level abstract data types that are built into the language, like classes, dictionaries, and lists. Since we're building up an understanding of how these abstractions work, we can't use them yet ourselves. As a consequence, we will write some code that isn't Pythonic --- it's not necessarily the typical way to implement our ideas in the language. What we write is

instructive, however, because it demonstrates how these abstractions can be constructed! Remember that computer science isn't just about learning to use programming languages, but also learning how they work.

2.2.1 Example: Arithmetic on Rational Numbers

Recall that a rational number is a ratio of integers, and rational numbers constitute an important sub-class of real numbers. A rational number like `1/3` or `17/29` is typically written as:

```
<numerator>/<denominator>
```

where both the `<numerator>` and `<denominator>` are placeholders for integer values. Both parts are needed to exactly characterize the value of the rational number.

Rational numbers are important in computer science because they, like integers, can be represented exactly. Irrational numbers (like `pi` or `e` or `sqrt(2)`) are instead approximated using a finite binary expansion. Thus, working with rational numbers should, in principle, allow us to avoid approximation errors in our arithmetic.

However, as soon as we actually divide the numerator by the denominator, we can be left with a truncated decimal approximation (a `float`).

```
>>> 1/3
0.3333333333333333
```

and the problems with this approximation appear when we start to conduct tests:

```
>>> 1/3 == 0.333333333333333300000  # Beware of approximations
True
```

How computers approximate real numbers with finite-length decimal expansions is a topic for another class. The important idea here is that by representing rational numbers as ratios of integers, we avoid the approximation problem entirely. Hence, we would like to keep the numerator and denominator separate for the sake of precision, but treat them as a single unit.

We know from using functional abstractions that we can start programming productively before we have an implementation of some parts of our program. Let us begin by assuming that we already have a way of constructing a rational number from a numerator and a

denominator. We also assume that, given a rational number, we have a way of extracting (or selecting) its numerator and its denominator. Let us further assume that the constructor and selectors are available as the following three functions:

- `make_rat(n, d)` returns the rational number with numerator `n` and denominator `d`.
- `numer(x)` returns the numerator of the rational number `x`.
- `denom(x)` returns the denominator of the rational number `x`.

We are using here a powerful strategy of synthesis: *wishful thinking*. We haven't yet said how a rational number is represented, or how the functions `numer`, `denom`, and `make_rat` should be implemented. Even so, if we did have these three functions, we could then add, multiply, and test equality of rational numbers by calling them:

```
>>> def add_rat(x, y):
        nx, dx = numer(x), denom(x)
        ny, dy = numer(y), denom(y)
        return make_rat(nx * dy + ny * dx, dx * dy)
```

```
>>> def mul_rat(x, y):
        return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

```
>>> def eq_rat(x, y):
        return numer(x) * denom(y) == numer(y) * denom(x)
```

Now we have the operations on rational numbers defined in terms of the selector functions `numer` and `denom`, and the constructor function `make_rat`, but we haven't yet defined these functions. What we need is some way to glue together a numerator and a denominator into a unit.

2.2.2 Tuples

To enable us to implement the concrete level of our data abstraction, Python provides a compound structure called a `tuple`, which can be constructed by separating values by commas. Although not strictly required, parentheses almost always surround tuples.

```
>>> (1, 2)
(1, 2)
```

The elements of a tuple can be unpacked in two ways. The first way is via our familiar method of multiple assignment.

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> x
1
>>> y
2
```

In fact, multiple assignment has been creating and unpacking tuples all along.

A second method for accessing the elements in a tuple is by the indexing operator, written as square brackets.

```
>>> pair[0]
1
>>> pair[1]
2
```

Tuples in Python (and sequences in most other programming languages) are 0-indexed, meaning that the index `0` picks out the first element, index `1` picks out the second, and so on. One intuition that underlies this indexing convention is that the index represents how far an element is offset from the beginning of the tuple.

The equivalent function for the element selection operator is called `getitem` , and it also uses 0-indexed positions to select elements from a tuple.

```
>>> from operator import getitem
>>> getitem(pair, 0)
1
```

Tuples are native types, which means that there are built-in Python operators to manipulate them. We'll return to the full properties of tuples shortly. At present, we are only interested in how tuples can serve as the glue that implements abstract data types.

Representing Rational Numbers. Tuples offer a natural way to implement rational numbers as a pair of two integers: a numerator and a denominator. We can implement our constructor and selector functions for rational numbers by manipulating 2-element tuples.

```
>>> def make_rat(n, d):
        return (n, d)
```

```
>>> def numer(x):
        return getitem(x, 0)
```

```
>>> def denom(x):
        return getitem(x, 1)
```

A function for printing rational numbers completes our implementation of this abstract data type.

```
>>> def str_rat(x):
        """Return a string 'n/d' for numerator n and denominator d."""
        return '{0}/{1}'.format(numer(x), denom(x))
```

Together with the arithmetic operations we defined earlier, we can manipulate rational numbers with the functions we have defined.

```
>>> half = make_rat(1, 2)
>>> str_rat(half)
'1/2'
>>> third = make_rat(1, 3)
>>> str_rat(mul_rat(half, third))
'1/6'
>>> str_rat(add_rat(third, third))
'6/9'
```

As the final example shows, our rational-number implementation does not reduce rational numbers to lowest terms. We can remedy this by changing `make_rat`. If we have a function for computing the greatest common denominator of two integers, we can use it to reduce the numerator and the denominator to lowest terms before constructing the pair. As with many useful tools, such a function already exists in the Python Library.

```
>>> from fractions import gcd
>>> def make_rat(n, d):
        g = gcd(n, d)
        return (n//g, d//g)
```

The double slash operator, `//`, expresses integer division, which rounds down the fractional part of the result of division. Since we know that `g` divides both `n` and `d` evenly, integer division is exact in this case. Now we have

```
>>> str_rat(add_rat(third, third))
'2/3'
```

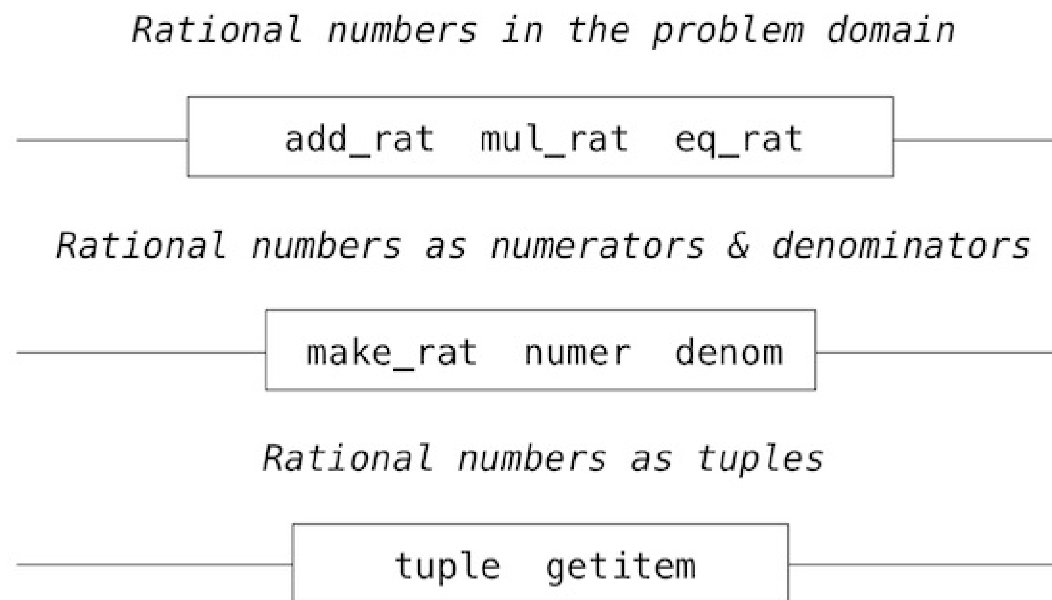
as desired. This modification was accomplished by changing the constructor without changing any of the functions that implement the actual arithmetic operations.

Further reading. The `str_rat` implementation above uses *format strings*, which contain placeholders for values. The details of how to use format strings and the `format` method appear in the formatting strings section of Dive Into Python 3.

2.2.3 Abstraction Barriers

Before continuing with more examples of compound data and data abstraction, let us consider some of the issues raised by the rational number example. We defined operations in terms of a constructor `make_rat` and selectors `numer` and `denom`. In general, the underlying idea of data abstraction is to identify for each type of value a basic set of operations in terms of which all manipulations of values of that type will be expressed, and then to use only those operations in manipulating the data.

We can envision the structure of the rational number system as a series of layers.



The horizontal lines represent abstraction barriers that isolate different levels of the system. At each level, the barrier separates the functions (above) that use the data abstraction from the functions (below) that implement the data abstraction. Programs that use rational numbers manipulate them solely in terms of the their arithmetic functions: `add_rat`, `mul_rat`, and `eq_rat`. These, in turn, are implemented solely in terms of the constructor and selectors `make_rat`, `numer`, and `denom`, which themselves are implemented in terms of tuples. The details of how tuples are implemented are irrelevant to the rest of the layers as long as tuples enable the implementation of the selectors and constructor.

At each layer, the functions within the box enforce the abstraction boundary because they are the only functions that depend upon both the representation above them (by their use) and the implementation below them (by their definitions). In this way, abstraction barriers are expressed as sets of functions.

Abstraction barriers provide many advantages. One advantage is that they makes programs much easier to maintain and to modify. The fewer functions that depend on a particular representation, the fewer changes are required when one wants to change that representation.

2.2.4 The Properties of Data

We began the rational-number implementation by implementing arithmetic operations in terms of three unspecified functions: `make_rat`, `numer`, and `denom`. At that point, we could think of the operations as being defined in terms of data objects --- numerators, denominators, and rational numbers --- whose behavior was specified by the latter three functions.

But what exactly is meant by data? It is not enough to say "whatever is implemented by the given selectors and constructors." We need to guarantee that these functions together specify the right behavior. That is, if we construct a rational number `x` from integers `n` and `d`, then it should be the case that `numer(x)/denom(x)` is equal to `n/d`.

In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

This point of view can be applied to other data types as well, such as the two-element tuple that we used in order to implement rational numbers. We never actually said much about what a tuple was, only that the language supplied operators to create and manipulate tuples. We can now describe the behavior conditions of two-element tuples, also called pairs, that are relevant to the problem of representing rational numbers.

In order to implement rational numbers, we needed a form of glue for two integers, which had the following behavior:

- If a pair `p` was constructed from values `x` and `y`, then `getitem_pair(p, 0)` returns `x`, and `getitem_pair(p, 1)` returns `y`.

We can implement functions `make_pair` and `getitem_pair` that fulfill this description just as well as a tuple.

```
>>> def make_pair(x, y):
        """Return a function that behaves like a pair."""
        def dispatch(m):
            if m == 0:
                return x
            elif m == 1:
                return y
        return dispatch
```

```
>>> def getitem_pair(p, i):
        """Return the element at index i of pair p."""
        return p(i)
```

With this implementation, we can create and manipulate pairs.

```
>>> p = make_pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
```

This use of functions corresponds to nothing like our intuitive notion of what data should be. Nevertheless, these functions suffice to represent compound data in our programs.

The subtle point to notice is that the value returned by `make_pair` is a function called `dispatch`, which takes an argument `m` and returns either `x` or `y`. Then, `getitem_pair` calls this function to retrieve the appropriate value. We will return to the topic of dispatch functions several times throughout this chapter.

The point of exhibiting the functional representation of a pair is not that Python actually works this way (tuples are implemented more directly, for efficiency reasons) but that it could work this way. The functional representation, although obscure, is a perfectly adequate way to represent pairs, since it fulfills the only conditions that pairs need to fulfill. This example also demonstrates that the ability to manipulate functions as values automatically provides us the ability to represent compound data.

2.3 Sequences

A sequence is an ordered collection of data values. Unlike a pair, which has exactly two elements, a sequence can have an arbitrary (but finite) number of ordered elements.

The sequence is a powerful, fundamental abstraction in computer science. For example, if we have sequences, we can list every student at Berkeley, or every university in the world, or every student in every university. We can list every class ever taken, every assignment ever completed, every grade ever received. The sequence abstraction enables the thousands of data-driven programs that impact our lives every day.

A sequence is not a particular abstract data type, but instead a collection of behaviors that different types share. That is, there are many kinds of sequences, but they all share certain properties. In particular,

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Unlike an abstract data type, we have not stated how to construct a sequence. The sequence abstraction is a collection of behaviors that does not fully specify a type (i.e., with constructors and selectors), but may be shared among several types. Sequences provide a layer of abstraction that may hide the details of exactly which sequence type is being manipulated by a particular program.

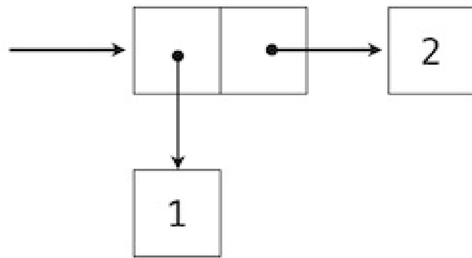
In this section, we develop a particular abstract data type that can implement the sequence abstraction. We then introduce built-in Python types that also implement the same abstraction.

2.3.1 Nested Pairs

For rational numbers, we paired together two integer objects using a two-element tuple, then showed that we could implement pairs just as well using functions. In that case, the elements of each pair we constructed were integers. However, like expressions, tuples can nest. Either element of a pair can itself be a pair, a property that holds true for either method of implementing a pair that we have seen: as a tuple or as a dispatch function.

A standard way to visualize a pair --- in this case, the pair `(1,2)` --- is called *box-and-pointer* notation. Each value, compound or primitive, is depicted as a pointer to a box. The box for a primitive value contains a representation of that value. For example, the box for a

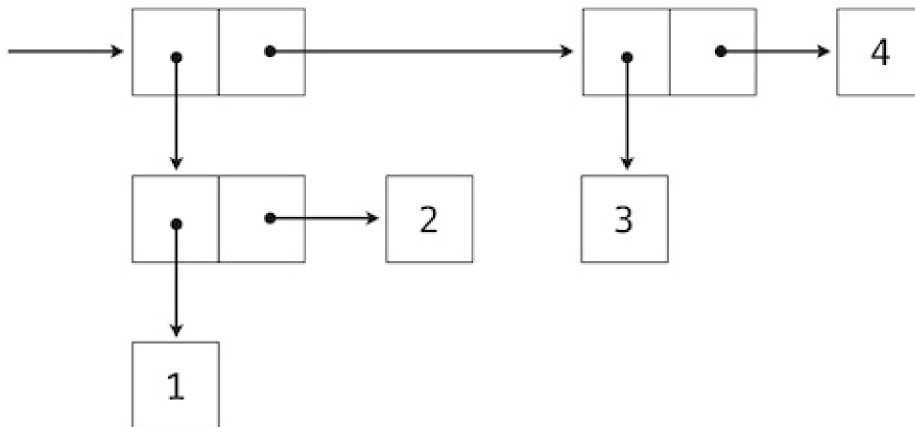
number contains a numeral. The box for a pair is actually a double box: the left part contains (an arrow to) the first element of the pair and the right part contains the second.



This Python expression for a nested tuple,

```
>>> ((1, 2), (3, 4))
((1, 2), (3, 4))
```

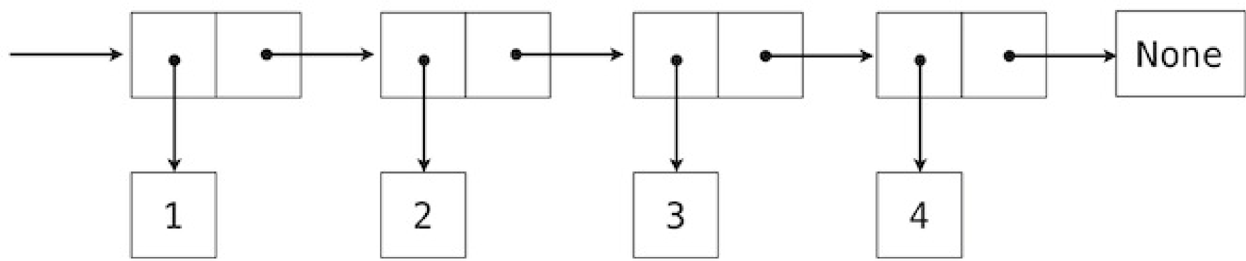
would have the following structure.



Our ability to use tuples as the elements of other tuples provides a new means of combination in our programming language. We call the ability for tuples to nest in this way a *closure property* of the tuple data type. In general, a method for combining data values satisfies the closure property if the result of combination can itself be combined using the same method. Closure is the key to power in any means of combination because it permits us to create hierarchical structures --- structures made up of parts, which themselves are made up of parts, and so on. We will explore a range of hierarchical structures in Chapter 3. For now, we consider a particularly important structure.

2.3.2 Recursive Lists

We can use nested pairs to form lists of elements of arbitrary length, which will allow us to implement the sequence abstraction. The figure below illustrates the structure of the recursive representation of a four-element list: `1`, `2`, `3`, `4`.



The list is represented by a chain of pairs. The first element of each pair is an element in the list, while the second is a pair that represents the rest of the list. The second element of the final pair is `None`, which indicates that the list has ended. We can construct this structure using a nested tuple literal:

```
>>> (1, (2, (3, (4, None))))
(1, (2, (3, (4, None))))
```

This nested structure corresponds to a very useful way of thinking about sequences in general, which we have seen before in the execution rules of the Python interpreter. A non-empty sequence can be decomposed into:

- its first element, and
- the rest of the sequence.

The rest of a sequence is itself a (possibly empty) sequence. We call this view of sequences recursive, because sequences contain other sequences as their second component.

Since our list representation is recursive, we will call it an `rlist` in our implementation, so as not to confuse it with the built-in `list` type in Python that we will introduce later in this chapter. A recursive list can be constructed from a first element and the rest of the list. The value `None` represents an empty recursive list.

```
>>> empty_rlist = None
>>> def make_rlist(first, rest):
        """Make a recursive list from its first element and the rest."""
        return (first, rest)
```

```
>>> def first(s):
        """Return the first element of a recursive list s."""
        return s[0]
```

```
>>> def rest(s):
        """Return the rest of the elements of a recursive list s."""
        return s[1]
```

These two selectors, one constructor, and one constant together implement the recursive list abstract data type. The single behavior condition for a recursive list is that, like a pair, its constructor and selectors are inverse functions.

- If a recursive list `s` was constructed from element `f` and list `r`, then `first(s)` returns `f`, and `rest(s)` returns `r`.

We can use the constructor and selectors to manipulate recursive lists.

```
>>> counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
>>> first(counts)
1
>>> rest(counts)
(2, (3, (4, None)))
```

Recall that we were able to represent pairs using functions, and therefore we can represent recursive lists using functions as well.

The recursive list can store a sequence of values in order, but it does not yet implement the sequence abstraction. Using the abstract data type we have defined, we can implement the two behaviors that characterize a sequence: length and element selection.

```
>>> def len_rlist(s):
        """Return the length of recursive list s."""
        length = 0
        while s != empty_rlist:
            s, length = rest(s), length + 1
        return length
```

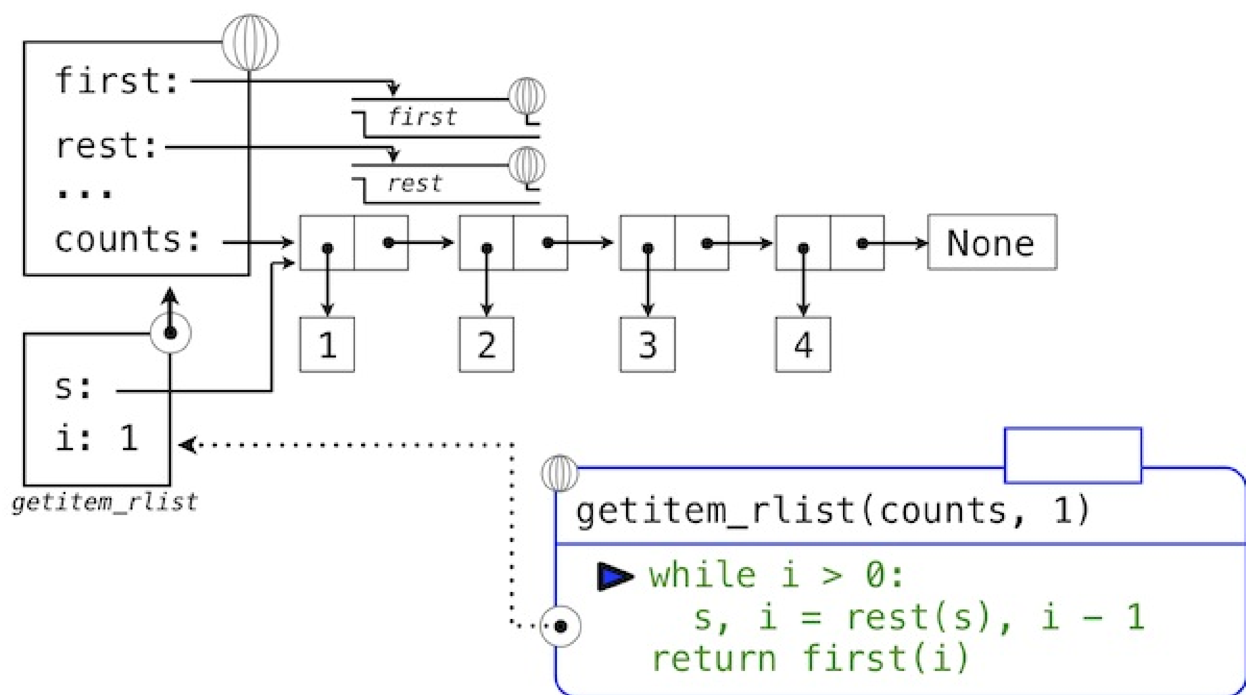
```
>>> def getitem_rlist(s, i):
        """Return the element at index i of recursive list s."""
        while i > 0:
            s, i = rest(s), i - 1
        return first(s)
```

Now, we can manipulate a recursive list as a sequence:

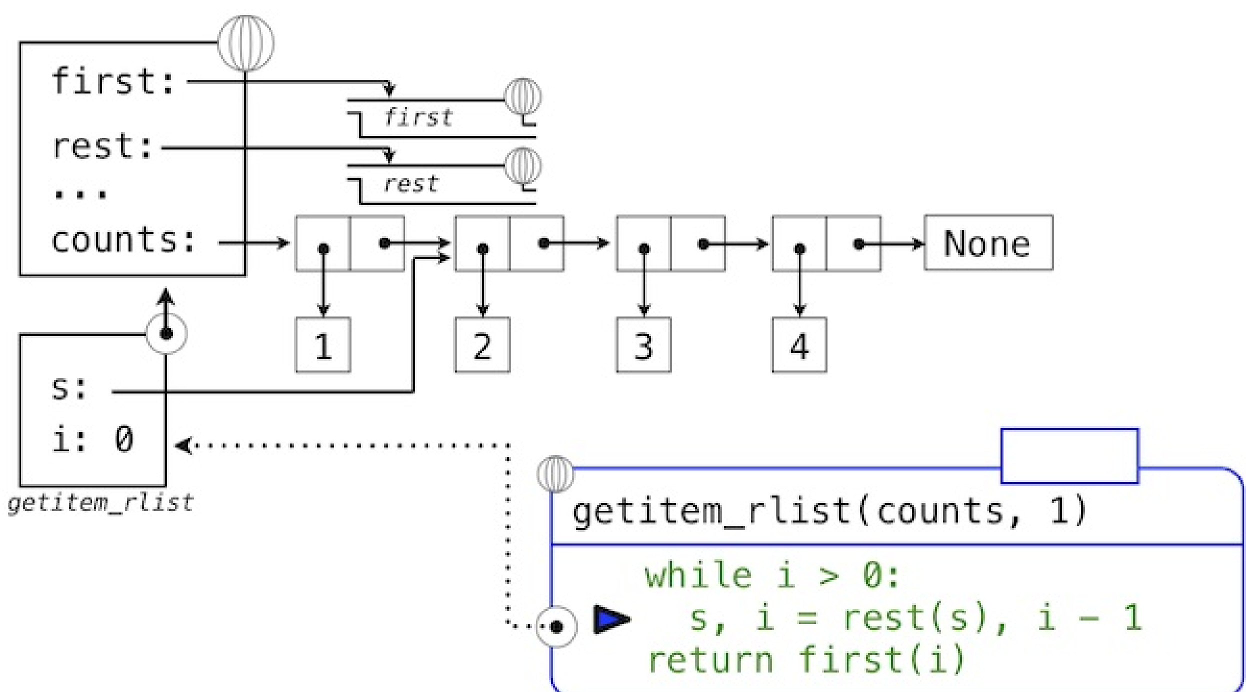
```
>>> len_rlist(counts)
4
>>> getitem_rlist(counts, 1)  # The second item has index 1
2
```

Both of these implementations are iterative. They peel away each layer of nested pair until the end of the list (in `len_rlist`) or the desired element (in `getitem_rlist`) is reached.

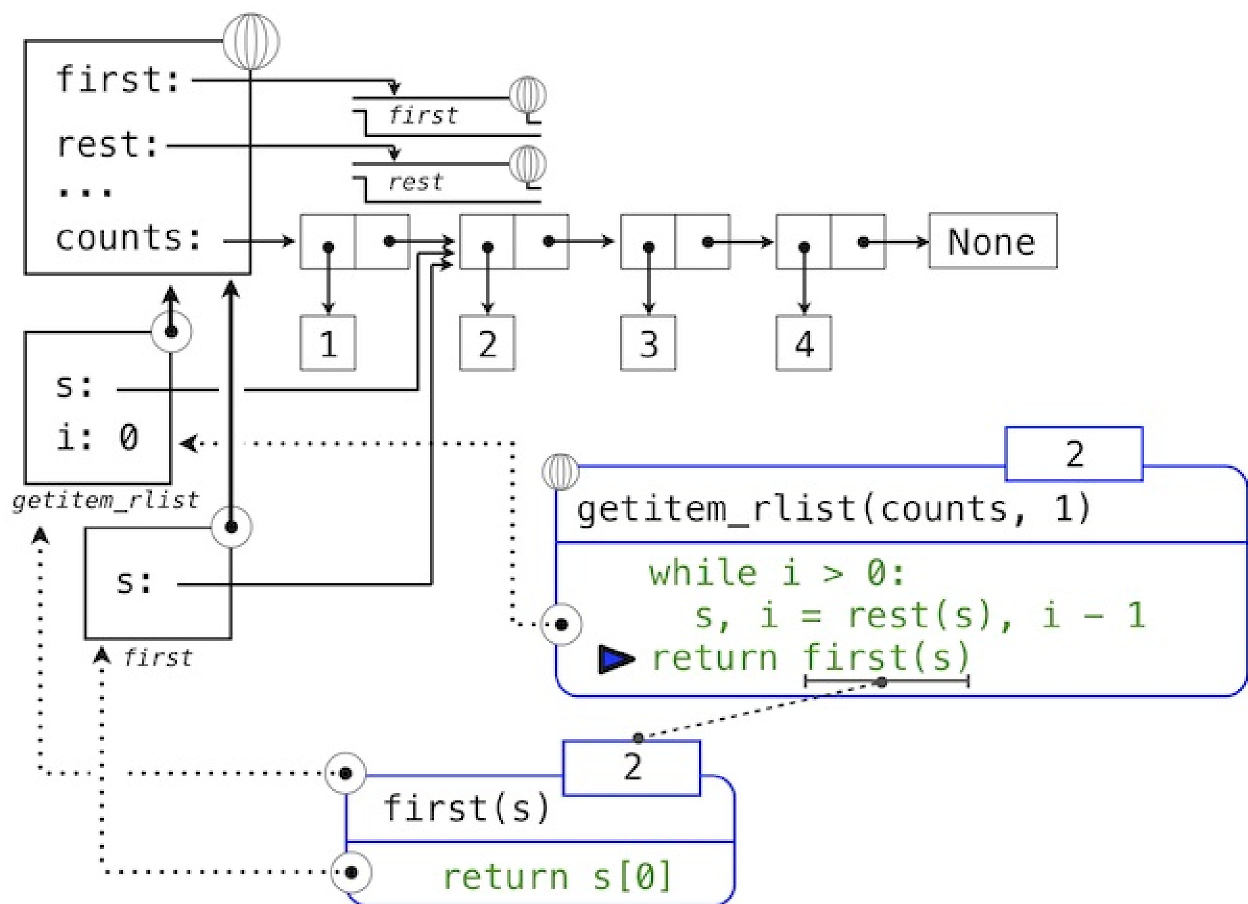
The series of environment diagrams below illustrate the iterative process by which `getitem_rlist` finds the element `2` at index `1` in the recursive list. First, the function `getitem_rlist` is called, creating a local frame.



The expression in the `while` header evaluates to true, which causes the assignment statement in the `while` suite to be executed.



In this case, the local name `s` now refers to the sub-list that begins with the second element of the original list. Evaluating the `while` header expression now yields a false value, and so Python evaluates the expression in the return statement on the final line of `getitem_rlist`.



This final environment diagram shows the local frame for the call to `first`, which contains the name `s` bound to that same sub-list. The `first` function selects the value `2` and returns it, completing the call to `getitem_rlist`.

This example demonstrates a common pattern of computation with recursive lists, where each step in an iteration operates on an increasingly shorter suffix of the original list. This incremental processing to find the length and elements of a recursive list does take some time to compute. (In Chapter 3, we will learn to characterize the computation time of iterative functions like these.) Python's built-in sequence types are implemented in a different way that does not have a large computational cost for computing the length of a sequence or retrieving its elements.

The way in which we construct recursive lists is rather verbose. Fortunately, Python provides a variety of built-in sequence types that provide both the versatility of the sequence abstraction, as well as convenient notation.

2.3.3 Tuples II

In fact, the `tuple` type that we introduced to form primitive pairs is itself a full sequence type. Tuples provide substantially more functionality than the pair abstract data type that we implemented functionally.

Tuples can have arbitrary length, and they exhibit the two principal behaviors of the sequence abstraction: length and element selection. Below, `digits` is a tuple with four elements.

```
>>> digits = (1, 8, 2, 8)
>>> len(digits)
4
>>> digits[3]
8
```

Additionally, tuples can be added together and multiplied by integers. For tuples, addition and multiplication do not add or multiply elements, but instead combine and replicate the tuples themselves. That is, the `add` function in the `operator` module (and the `+` operator) returns a new tuple that is the conjunction of the added arguments. The `mul` function in `operator` (and the `*` operator) can take an integer `k` and a tuple and return a new tuple that consists of `k` copies of the tuple argument.

```
>>> (2, 7) + digits * 2
(2, 7, 1, 8, 2, 8, 1, 8, 2, 8)
```

Mapping. A powerful method of transforming one tuple into another is by applying a function to each element and collecting the results. This general form of computation is called *mapping* a function over a sequence, and corresponds to the built-in function `map`. The result of `map` is an object that is not itself a sequence, but can be converted into a sequence by calling `tuple`, the constructor function for tuples.

```
>>> alternates = (-1, 2, -3, 4, -5)
>>> tuple(map(abs, alternates))
(1, 2, 3, 4, 5)
```

The `map` function is important because it relies on the sequence abstraction: we do not need to be concerned about the structure of the underlying tuple; only that we can access each one of its elements individually in order to pass it as an argument to the mapped function (`abs`, in this case).

2.3.4 Sequence Iteration

Mapping is itself an instance of a general pattern of computation: iterating over all elements in a sequence. To map a function over a sequence, we do not just select a particular element, but each element in turn. This pattern is so common that Python has an additional control statement to process sequential data: the `for` statement.

Consider the problem of counting how many times a value appears in a sequence. We can implement a function to compute this count using a `while` loop.

```
>>> def count(s, value):
        """Count the number of occurrences of value in sequence s."""
        total, index = 0, 0
        while index < len(s):
            if s[index] == value:
                total = total + 1
            index = index + 1
        return total
```

```
>>> count(digits, 8)
2
```

The Python `for` statement can simplify this function body by iterating over the element values directly, without introducing the name `index` at all. `For` example (pun intended), we can write:

```
>>> def count(s, value):
        """Count the number of occurrences of value in sequence s."""
        total = 0
        for elem in s:
            if elem == value:
                total = total + 1
        return total
```

```
>>> count(digits, 8)
2
```

A `for` statement consists of a single clause with the form:

```
for <name> in <expression>:
    <suite>
```

A `for` statement is executed by the following procedure:

1. Evaluate the header `<expression>`, which must yield an iterable value.
2. For each element value in that sequence, in order:
    i. Bind `<name>` to that value in the local environment.
    ii. Execute the `<suite>`.

Step 1 refers to an iterable value. Sequences are iterable, and their elements are considered in their sequential order. Python does include other iterable types, but we will focus on sequences for now; the general definition of the term "iterable" appears in the section on iterators in Chapter 4.

An important consequence of this evaluation procedure is that `<name>` will be bound to the last element of the sequence after the `for` statement is executed. The `for` loop introduces yet another way in which the local environment can be updated by a statement.

Sequence unpacking. A common pattern in programs is to have a sequence of elements that are themselves sequences, but all of a fixed length. `For` statements may include multiple names in their header to "unpack" each element sequence into its respective elements. For example, we may have a sequence of pairs (that is, two-element tuples),

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))
```

and wish to find the number of pairs that have the same first and second element.

```
>>> same_count = 0
```

The following `for` statement with two names in its header will bind each name `x` and `y` to the first and second elements in each pair, respectively.

```
>>> for x, y in pairs:
        if x == y:
            same_count = same_count + 1
```

```
>>> same_count
2
```

This pattern of binding multiple names to multiple values in a fixed-length sequence is called *sequence unpacking*; it is the same pattern that we see in assignment statements that bind multiple names to multiple values.

Ranges. A `range` is another built-in type of sequence in Python, which represents a range of integers. Ranges are created with the `range` function, which takes two integer arguments: the first number and one beyond the last number in the desired range.

```
>>> range(1, 10)  # Includes 1, but not 10
range(1, 10)
```

Calling the `tuple` constructor on a range will create a tuple with the same elements as the range, so that the elements can be easily inspected.

```
>>> tuple(range(5, 8))
(5, 6, 7)
```

If only one argument is given, it is interpreted as one beyond the last value for a range that starts at 0.

```
>>> tuple(range(4))
(0, 1, 2, 3)
```

Ranges commonly appear as the expression in a `for` header to specify the number of times that the suite should be executed:

```
>>> total = 0
>>> for k in range(5, 8):
        total = total + k
```

```
>>> total
18
```

A common convention is to use a single underscore character for the name in the `for` header if the name is unused in the suite:

```
>>> for _ in range(3):
        print('Go Bears!')

Go Bears!
Go Bears!
Go Bears!
```

Note that an underscore is just another name in the environment as far as the interpreter is concerned, but has a conventional meaning among programmers that indicates the name will not appear in any expressions.

2.3.5 Sequence Abstraction

We have now introduced two types of native data types that implement the sequence abstraction: tuples and ranges. Both satisfy the conditions with which we began this section: length and element selection. Python includes two more behaviors of sequence types that extend the sequence abstraction.

Membership. A value can be tested for membership in a sequence. Python has two operators `in` and `not in` that evaluate to `True` or `False` depending on whether an element appears in a sequence.

```
>>> digits
(1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

All sequences also have methods called `index` and `count`, which return the index of (or count of) a value in a sequence.

Slicing. Sequences contain smaller sequences within them. We observed this property when developing our nested pairs implementation, which decomposed a sequence into its first element and the rest. A *slice* of a sequence is any span of the original sequence, designated by a pair of integers. As with the `range` constructor, the first integer indicates the starting index of the slice and the second indicates one beyond the ending index.

In Python, sequence slicing is expressed similarly to element selection, using square brackets. A colon separates the starting and ending indices. Any bound that is omitted is assumed to be an extreme value: 0 for the starting index, and the length of the sequence for the ending index.

```
>>> digits[0:2]
(1, 8)
>>> digits[1:]
(8, 2, 8)
```

Enumerating these additional behaviors of the Python sequence abstraction gives us an opportunity to reflect upon what constitutes a useful data abstraction in general. The richness of an abstraction (that is, how many behaviors it includes) has consequences. For users of an abstraction, additional behaviors can be helpful. On the other hand, satisfying the requirements of a rich abstraction with a new data type can be challenging. To ensure that our implementation of recursive lists supported these additional behaviors would require some work. Another negative consequence of rich abstractions is that they take longer for users to learn.

Sequences have a rich abstraction because they are so ubiquitous in computing that learning a few complex behaviors is justified. In general, most user-defined abstractions should be kept as simple as possible.

Further reading. Slice notation admits a variety of special cases, such as negative starting values, ending values, and step sizes. A complete description appears in the subsection called slicing a list in Dive Into Python 3. In this chapter, we will only use the basic features described above.

2.3.6 Strings

Text values are perhaps more fundamental to computer science than even numbers. As a case in point, Python programs are written and stored as text. The native data type for text in Python is called a string, and corresponds to the constructor `str`.

There are many details of how strings are represented, expressed, and manipulated in Python. Strings are another example of a rich abstraction, one which requires a substantial commitment on the part of the programmer to master. This section serves as a condensed introduction to essential string behaviors.

String literals can express arbitrary text, surrounded by either single or double quotation marks.

```
>>> 'I am string!'
'I am string!'
>>> "I've got an apostrophe"
"I've got an apostrophe"
>>> '您好'
'您好'
```

We have seen strings already in our code, as docstrings, in calls to `print`, and as error messages in `assert` statements.

Strings satisfy the two basic conditions of a sequence that we introduced at the beginning of this section: they have a length and they support element selection.

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

The elements of a string are themselves strings that have only a single character. A character is any single letter of the alphabet, punctuation mark, or other symbol. Unlike many other programming languages, Python does not have a separate character type; any text is a string, and strings that represent single characters have a length of 1.

Like tuples, strings can also be combined via addition and multiplication.

```
>>> 'Berkeley' + ', CA'
'Berkeley, CA'
>>> 'Shabu ' * 2
'Shabu Shabu '
```

Membership. The behavior of strings diverges from other sequence types in Python. The string abstraction does not conform to the full sequence abstraction that we described for tuples and ranges. In particular, the membership operator `in` applies to strings, but has an entirely different behavior than when it is applied to sequences. It matches substrings rather than elements.

```
>>> 'here' in "Where's Waldo?"
True
```

Likewise, the `count` and `index` methods on strings take substrings as arguments, rather than single-character elements. The behavior of `count` is particularly nuanced; it counts the number of non-overlapping occurrences of a substring in a string.

```
>>> 'Mississippi'.count('i')
4
>>> 'Mississippi'.count('issi')
1
```

Multiline Literals. Strings aren't limited to a single line. Triple quotes delimit string literals that span multiple lines. We have used this triple quoting extensively already for docstrings.

```
>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, "Readability counts."\nRead more: import this.'
```

In the printed result above, the `\n` (pronounced "*backslash en*") is a single element that represents a new line. Although it appears as two characters (backslash and "n"), it is considered a single character for the purposes of length and element selection.

String Coercion. A string can be created from any object in Python by calling the `str` constructor function with an object value as its argument. This feature of strings is useful for constructing descriptive strings from objects of various types.

```
>>> str(2) + ' is an element of ' + str(digits)
'2 is an element of (1, 8, 2, 8)'
```

The mechanism by which a single `str` function can apply to any type of argument and return an appropriate value is the subject of the later section on generic functions.

Methods. The behavior of strings in Python is extremely productive because of a rich set of methods for returning string variants and searching for contents. A few of these methods are introduced below by example.

```
>>> '1234'.isnumeric()
True
>>> 'rOBERT dE nIRO'.swapcase()
'Robert De Niro'
>>> 'snakeyes'.upper().endswith('YES')
True
```

Further reading. Encoding text in computers is a complex topic. In this chapter, we will abstract away the details of how strings are represented. However, for many applications, the particular details of how strings are encoded by computers is essential knowledge. Sections 4.1-4.3 of Dive Into Python 3 provides a description of character encodings and Unicode.

2.3.7 Conventional Interfaces

In working with compound data, we've stressed how data abstraction permits us to design programs without becoming enmeshed in the details of data representations, and how abstraction preserves for us the flexibility to experiment with alternative representations. In this section, we introduce another powerful design principle for working with data structures --- the use of *conventional interfaces*.

A conventional interface is a data format that is shared across many modular components, which can be mixed and matched to perform data processing. For example, if we have several functions that all take a sequence as an argument and return a sequence as a value, then we can apply each to the output of the next in any order we choose. In this way, we can create a complex process by chaining together a pipeline of functions, each of which is simple and focused.

This section has a dual purpose: to introduce the idea of organizing a program around a conventional interface, and to demonstrate examples of modular sequence processing.

Consider these two problems, which appear at first to be related only in their use of sequences:

1. Sum the even members of the first `n` Fibonacci numbers.
2. List the letters in the acronym for a name, which includes the first letter of each capitalized word.

These problems are related because they can be decomposed into simple operations that take sequences as input and yield sequences as output. Moreover, those operations are instances of general methods of computation over sequences. Let's consider the first problem. It can be decomposed into the following steps:

```
 enumerate     map    filter  accumulate
-----------    ---    ------  ----------
naturals(n)    fib    iseven     sum
```

The `fib` function below computes Fibonacci numbers (now updated from the definition in Chapter 1 with a `for` statement),

```
>>> def fib(k):
        """Compute the kth Fibonacci number."""
        prev, curr = 1, 0  # curr is the first Fibonacci number.
        for _ in range(k - 1):
             prev, curr = curr, prev + curr
        return curr
```

and a predicate `iseven` can be defined using the integer remainder operator, `%` .

```
>>> def iseven(n):
        return n % 2 == 0
```

The functions `map` and `filter` are operations on sequences. We have already encountered `map` , which applies a function to each element in a sequence and collects the results. The `filter` function takes a sequence and returns those elements of a sequence for which a predicate is true. Both of these functions return intermediate objects, `map` and `filter` objects, which are iterable objects that can be converted into tuples or summed.

```
>>> nums = (5, 6, -7, -8, 9)
>>> tuple(filter(iseven, nums))
(6, -8)
>>> sum(map(abs, nums))
35
```

Now we can implement `even_fib` , the solution to our first problem, in terms of `map` , `filter` , and `sum` .

```
>>> def sum_even_fibs(n):
        """Sum the first n even Fibonacci numbers."""
        return sum(filter(iseven, map(fib, range(1, n+1))))
```

```
>>> sum_even_fibs(20)
3382
```

Now, let's consider the second problem. It can also be decomposed as a pipeline of sequence operations that include `map` and `filter` :

```
enumerate  filter   map   accumulate
---------  ------  -----  ----------
  words    iscap   first    tuple
```

The words in a string can be enumerated via the `split` method of a string object, which by default splits on spaces.

```
>>> tuple('Spaces between words'.split())
('Spaces', 'between', 'words')
```

The first letter of a word can be retrieved using the selection operator, and a predicate that determines if a word is capitalized can be defined using the built-in predicate `isupper`.

```
>>> def first(s):
        return s[0]
```

```
>>> def iscap(s):
        return len(s) > 0 and s[0].isupper()
```

At this point, our acronym function can be defined via `map` and `filter`.

```
>>> def acronym(name):
        """Return a tuple of the letters that form the acronym for name."""
        return tuple(map(first, filter(iscap, name.split())))
```

```
>>> acronym('University of California Berkeley Undergraduate Graphics Group')
('U', 'C', 'B', 'U', 'G', 'G')
```

These similar solutions to rather different problems show how to combine general components that operate on the conventional interface of a sequence using the general computational patterns of mapping, filtering, and accumulation. The sequence abstraction allows us to specify these solutions concisely.

Expressing programs as sequence operations helps us design programs that are modular. That is, our designs are constructed by combining relatively independent pieces, each of which transforms a sequence. In general, we can encourage modular design by providing a library of standard components together with a conventional interface for connecting the components in flexible ways.

Generator expressions. The Python language includes a second approach to processing sequences, called *generator expressions*. which provide similar functionality to `map` and `filter`, but may require fewer function definitions.

Generator expressions combine the ideas of filtering and mapping together into a single expression type with the following form:

```
<map expression> for <name> in <sequence expression> if <filter expression>
```

To evaluate a generator expression, Python evaluates the `<sequence expression>`, which must return an iterable value. Then, for each element in order, the element value is bound to `<name>`, the filter expression is evaluated, and if it yields a true value, the map expression is evaluated.

The result value of evaluating a generator expression is itself an iterable value. Accumulation functions like `tuple`, `sum`, `max`, and `min` can take this returned object as an argument.

```
>>> def acronym(name):
        return tuple(w[0] for w in name.split() if iscap(w))
```

```
>>> def sum_even_fibs(n):
        return sum(fib(k) for k in range(1, n+1) if fib(k) % 2 == 0)
```

Generator expressions are specialized syntax that utilizes the conventional interface of iterable values, such as sequences. These expressions subsume most of the functionality of `map` and `filter`, but avoid actually creating the function values that are applied (or, incidentally, creating the environment frames required to apply those functions).

Reduce. In our examples we used specific functions to accumulate results, either `tuple` or `sum`. Functional programming languages (including Python) include general higher-order accumulators that go by various names. Python includes `reduce` in the `functools` module, which applies a two-argument function cumulatively to the elements of a sequence from left to right, to reduce a sequence to a value. The following expression computes 5 factorial.

```
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5))
120
```

Using this more general form of accumulation, we can also compute the product of even Fibonacci numbers, in addition to the sum, using sequences as a conventional interface.

```
>>> def product_even_fibs(n):
        """Return the product of the first n even Fibonacci numbers, except 0."""
        return reduce(mul, filter(iseven, map(fib, range(2, n+1))))
```

```
>>> product_even_fibs(20)
123476336640
```

The combination of higher order procedures corresponding to `map`, `filter`, and `reduce` will appear again in Chapter 4, when we consider methods for distributing computation across multiple computers.

2.4 Mutable Data

We have seen how abstraction is vital in helping us to cope with the complexity of large systems. Effective program synthesis also requires organizational principles that can guide us in formulating the overall design of a program. In particular, we need strategies to help us structure large systems so that they will be *modular*, that is, so that they can be divided "naturally" into coherent parts that can be separately developed and maintained.

One powerful technique for creating modular programs is to introduce new kinds of data that may change state over time. In this way, a single data object can represent something that evolves independently of the rest of the program. The behavior of a changing object may be influenced by its history, just like an entity in the world. Adding state to data is an essential ingredient of our final destination in this chapter: object-oriented programming.

The native data types we have introduced so far --- numbers, Booleans, tuples, ranges, and strings --- are all types of *immutable* objects. While names may change bindings to different values in the environment during the course of execution, the values themselves do not change. In this section, we will introduce a collection of *mutable* data types. Mutable objects can change throughout the execution of a program.

2.4.1 Local State

Our first example of a mutable object will be a function that has local state. That state will change during the course of execution of a program.

To illustrate what we mean by having a function with local state, let us model the situation of withdrawing money from a bank account. We will do so by creating a function called `withdraw`, which takes as its argument an amount to be withdrawn. If there is enough money in the account to accommodate the withdrawal, then `withdraw` should return the balance remaining after the withdrawal. Otherwise, `withdraw` should return the message `'Insufficient funds'`. For example, if we begin with $100 in the account, we would like to obtain the following sequence of return values by calling withdraw:

```
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'Insufficient funds'
>>> withdraw(15)
35
```

Observe that the expression `withdraw(25)`, evaluated twice, yields different values. This is a new kind of behavior for a user-defined function: it is non-pure. Calling the function not only returns a value, but also has the side effect of changing the function in some way, so that the next call with the same argument will return a different result. All of our user-defined functions so far have been pure functions, unless they called a non-pure built-in function. They have remained pure because they have not been allowed to make any changes outside of their local environment frame!

For `withdraw` to make sense, it must be created with an initial account balance. The function `make_withdraw` is a higher-order function that takes a starting balance as an argument. The function `withdraw` is its return value.

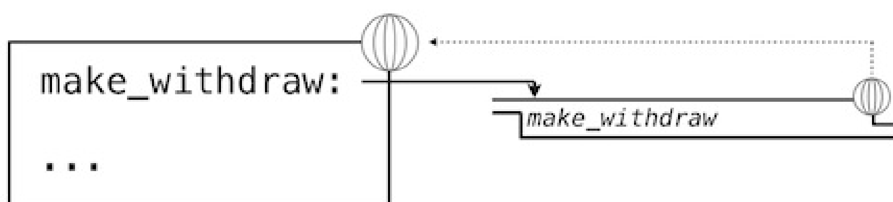
```
>>> withdraw = make_withdraw(100)
```

An implementation of `make_withdraw` requires a new kind of statement: a `nonlocal` statement. When we call `make_withdraw`, we bind the name `balance` to the initial amount. We then define and return a local function, `withdraw`, which updates and returns the value of `balance` when called.

```
>>> def make_withdraw(balance):
        """Return a withdraw function that draws down balance with each call."""
        def withdraw(amount):
            nonlocal balance                 # Declare the name "balance" nonlocal
            if amount > balance:
                return 'Insufficient funds'
            balance = balance - amount       # Re-bind the existing balance name
            return balance
        return withdraw
```

The novel part of this implementation is the `nonlocal` statement, which mandates that whenever we change the binding of the name `balance`, the binding is changed in the first frame in which `balance` is already bound. Recall that without the `nonlocal` statement, an assignment statement would always bind a name in the first frame of the environment. The `nonlocal` statement indicates that the name appears somewhere in the environment other than the first (local) frame or the last (global) frame.

We can visualize these changes with environment diagrams. The following environment diagrams illustrate the effects of each call, starting with the definition above. We abbreviate away code in the function values and expression trees that isn't central to our discussion.

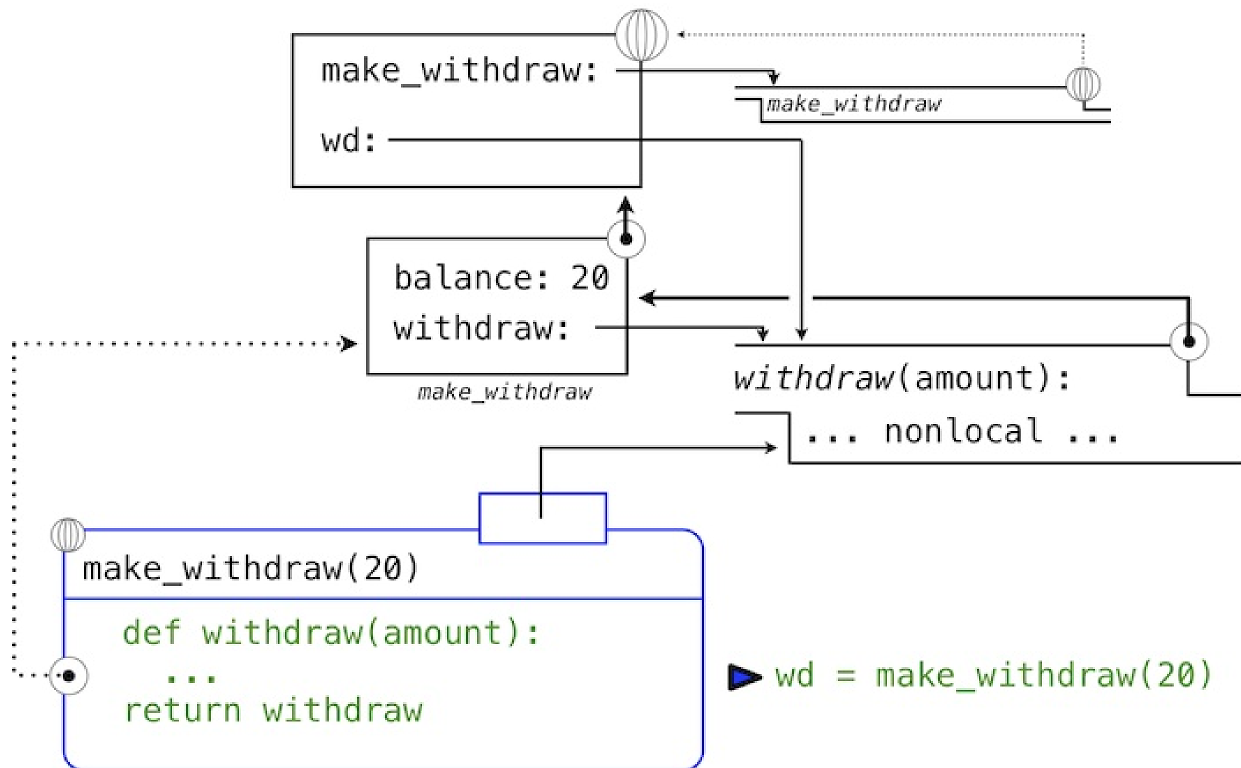


Our definition statement has the usual effect: it creates a new user-defined function and binds the name `make_withdraw` to that function in the global frame.

Next, we call `make_withdraw` with an initial balance argument of `20`.

```
>>> wd = make_withdraw(20)
```

This assignment statement binds the name `wd` to the returned function in the global frame.

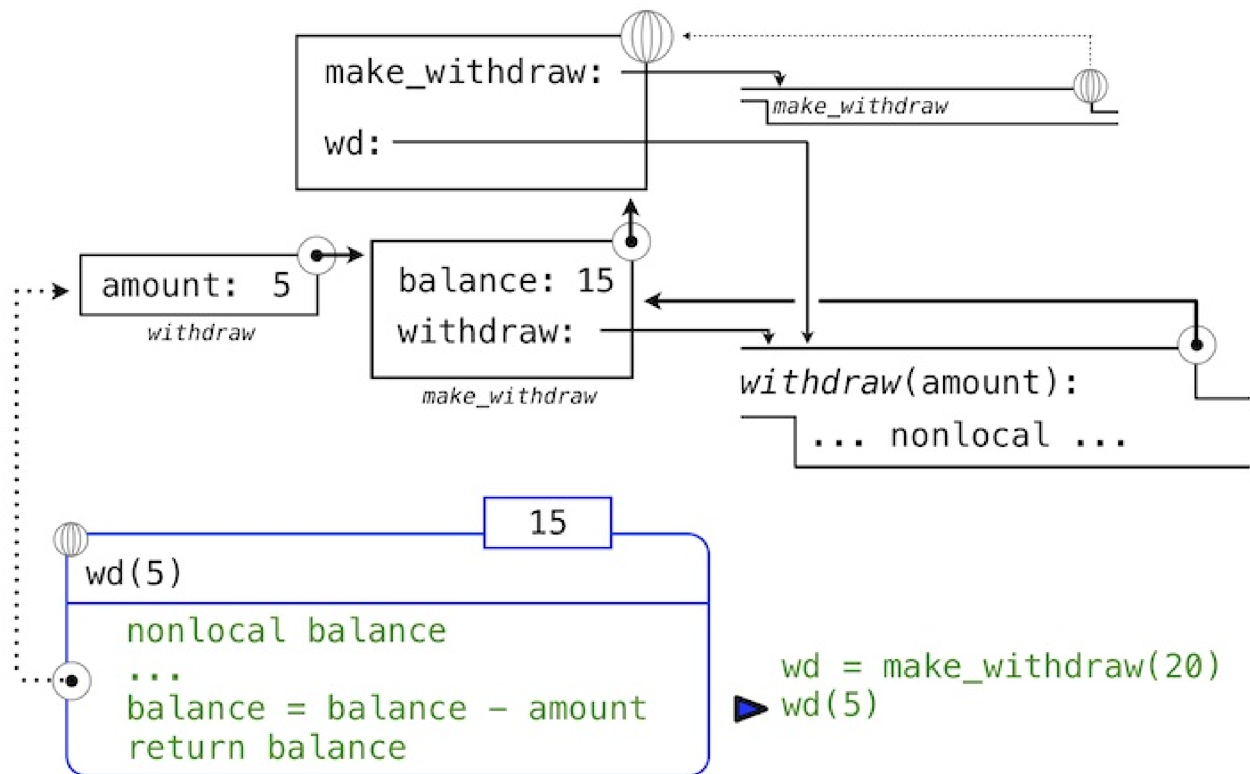


The returned function, (intrinsically) called *withdraw*, is associated with the local environment for the *make_withdraw* invocation in which it was defined. The name `balance` is bound in this local environment. Crucially, there will only be this single binding for the name `balance` throughout the rest of this example.

Next, we evaluate an expression that calls *withdraw* on an amount `5`.

```
>>> wd(5)
15
```

The name `wd` is bound to the *withdraw* function, so the body of *withdraw* is evaluated in a new environment that extends the environment in which *withdraw* was defined. Tracing the effect of evaluating *withdraw* illustrates the effect of a `nonlocal` statement in Python.



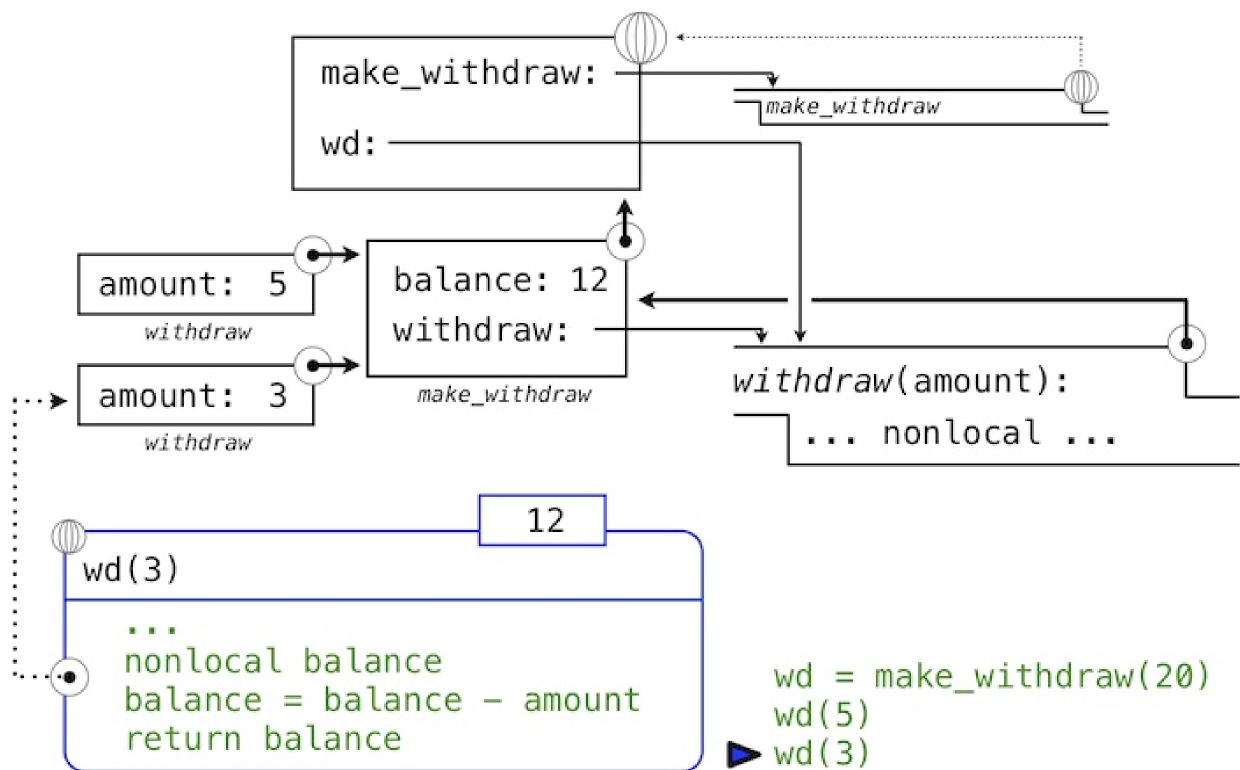
The assignment statement in *withdraw* would normally create a new binding for `balance` in *withdraw*'s local frame. Instead, because of the `nonlocal` statement, the assignment finds the first frame in which `balance` was already defined, and it rebinds the name in that frame. If `balance` had not previously been bound to a value, then the `nonlocal` statement would have given an error.

By virtue of changing the binding for `balance`, we have changed the *withdraw* function as well. The next time *withdraw* is called, the name `balance` will evaluate to `15` instead of `20`.

When we call `wd` a second time,

```
>>> wd(3)
12
```

we see that the changes to the value bound to the name `balance` are cumulative across the two calls.



Here, the second call to *withdraw* did create a second local frame, as usual. However, both *withdraw* frames extend the environment for *make_withdraw*, which contains the binding for `balance`. Hence, they share that particular name binding. Calling *withdraw* has the side effect of altering the environment that will be extended by future calls to *withdraw*.

Practical guidance. By introducing `nonlocal` statements, we have created a dual role for assignment statements. Either they change local bindings, or they change nonlocal bindings. In fact, assignment statements already had a dual role: they either created new bindings or re-bound existing names. The many roles of Python assignment can obscure the effects of executing an assignment statement. It is up to you as a programmer to document your code clearly so that the effects of assignment can be understood by others.

2.4.2 The Benefits of Non-Local Assignment

Non-local assignment is an important step on our path to viewing a program as a collection of independent and autonomous *objects*, which interact with each other but each manage their own internal state.

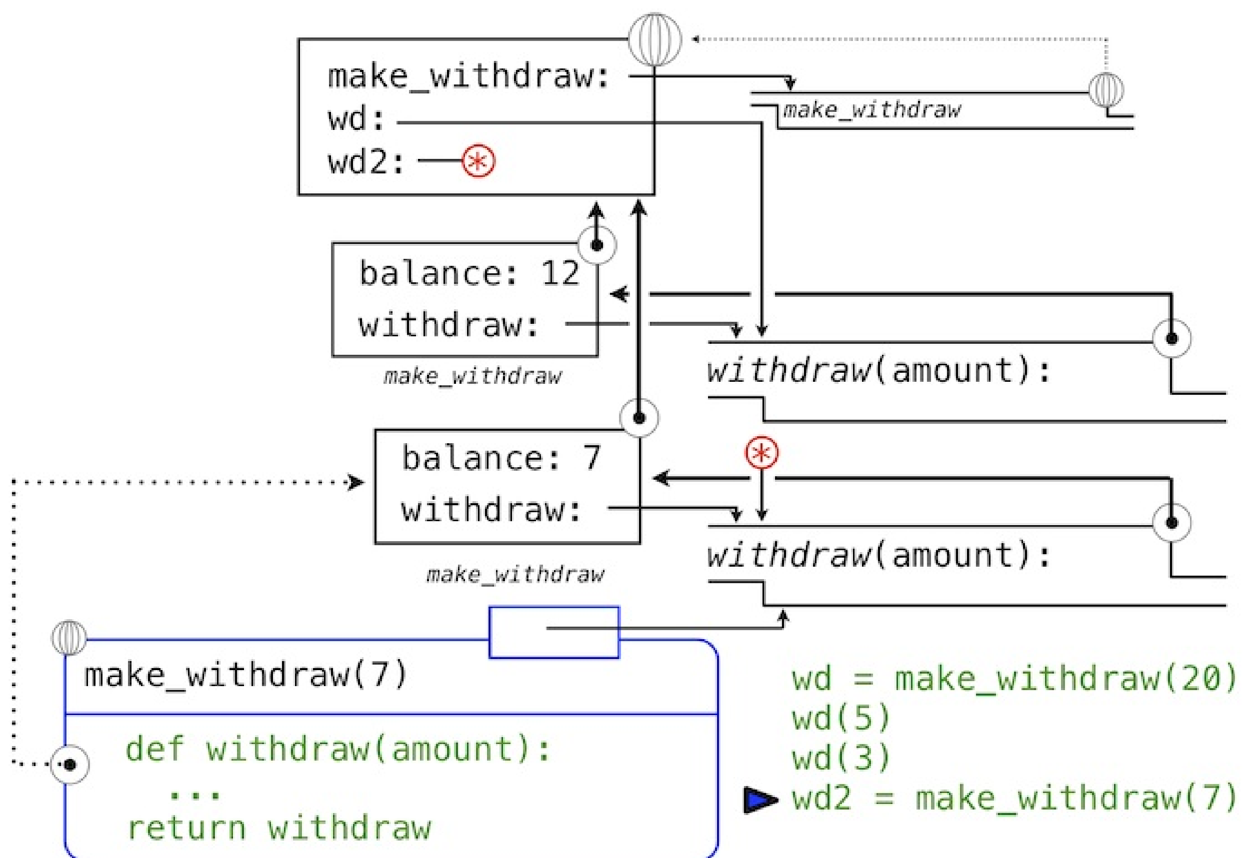
In particular, non-local assignment has given us the ability to maintain some state that is local to a function, but evolves over successive calls to that function. The `balance` associated with a particular withdraw function is shared among all calls to that function. However, the binding for balance associated with an instance of withdraw is inaccessible to

the rest of the program. Only *withdraw* is associated with the frame for *make_withdraw* in which it was defined. If *make_withdraw* is called again, then it will create a separate frame with a separate binding for `balance`.

We can continue our example to illustrate this point. A second call to *make_withdraw* returns a second *withdraw* function that is associated with yet another environment.

```
>>> wd2 = make_withdraw(7)
```

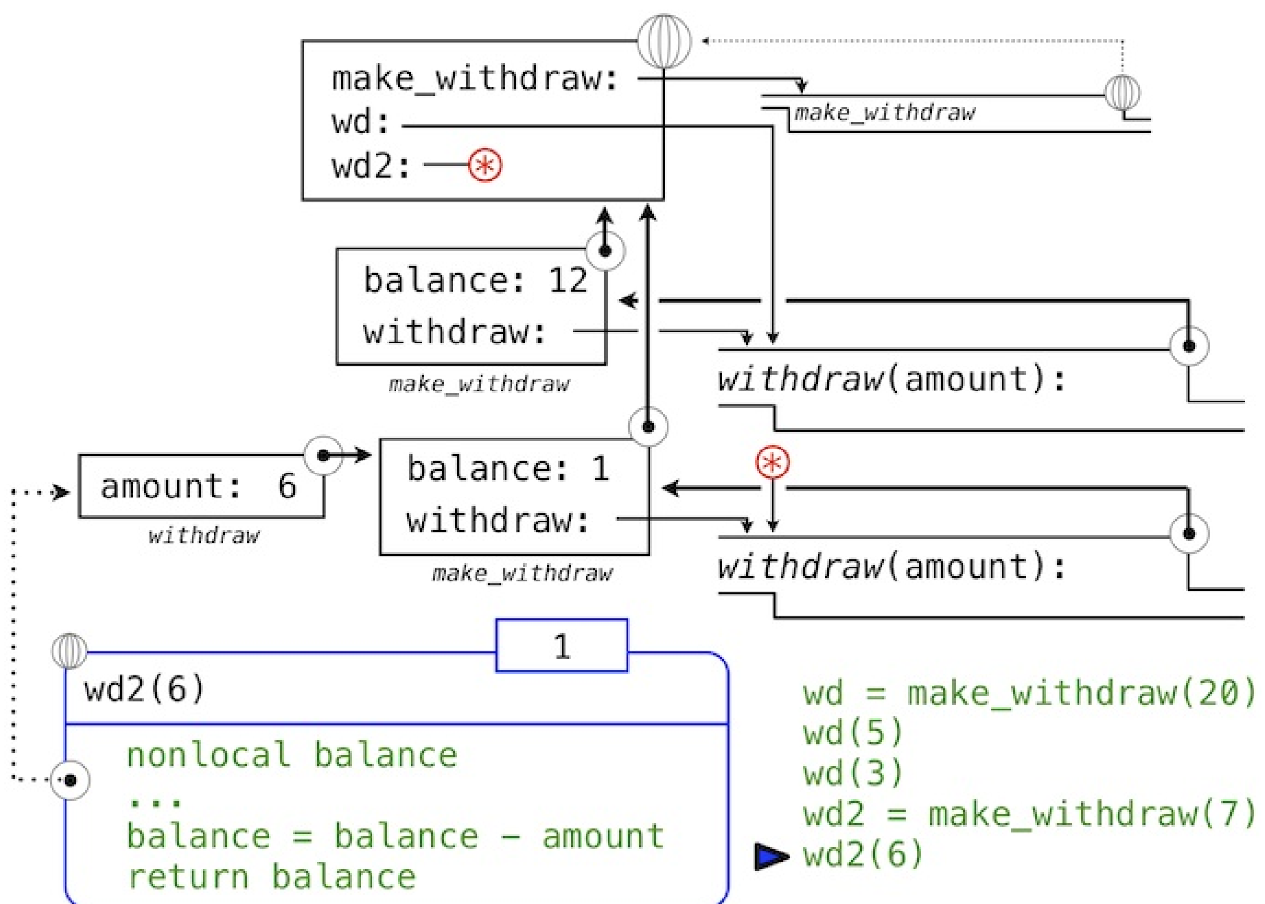
This second *withdraw* function is bound to the name `wd2` in the global frame. We've abbreviated the line that represents this binding with an asterisk. Now, we see that there are in fact two bindings for the name `balance`. The name `wd` is still bound to a *withdraw* function with a balance of `12`, while `wd2` is bound to a new *withdraw* function with a balance of `7`.



Finally, we call the second *withdraw* bound to `wd2`:

```
>>> wd2(6)
1
```

This call changes the binding of its nonlocal `balance` name, but does not affect the first *withdraw* bound to the name `wd` in the global frame.



In this way, each instance of *withdraw* is maintaining its own balance state, but that state is inaccessible to any other function in the program. Viewing this situation at a higher level, we have created an abstraction of a bank account that manages its own internals but behaves in a way that models accounts in the world: it changes over time based on its own history of withdrawal requests.

2.4.3 The Cost of Non-Local Assignment

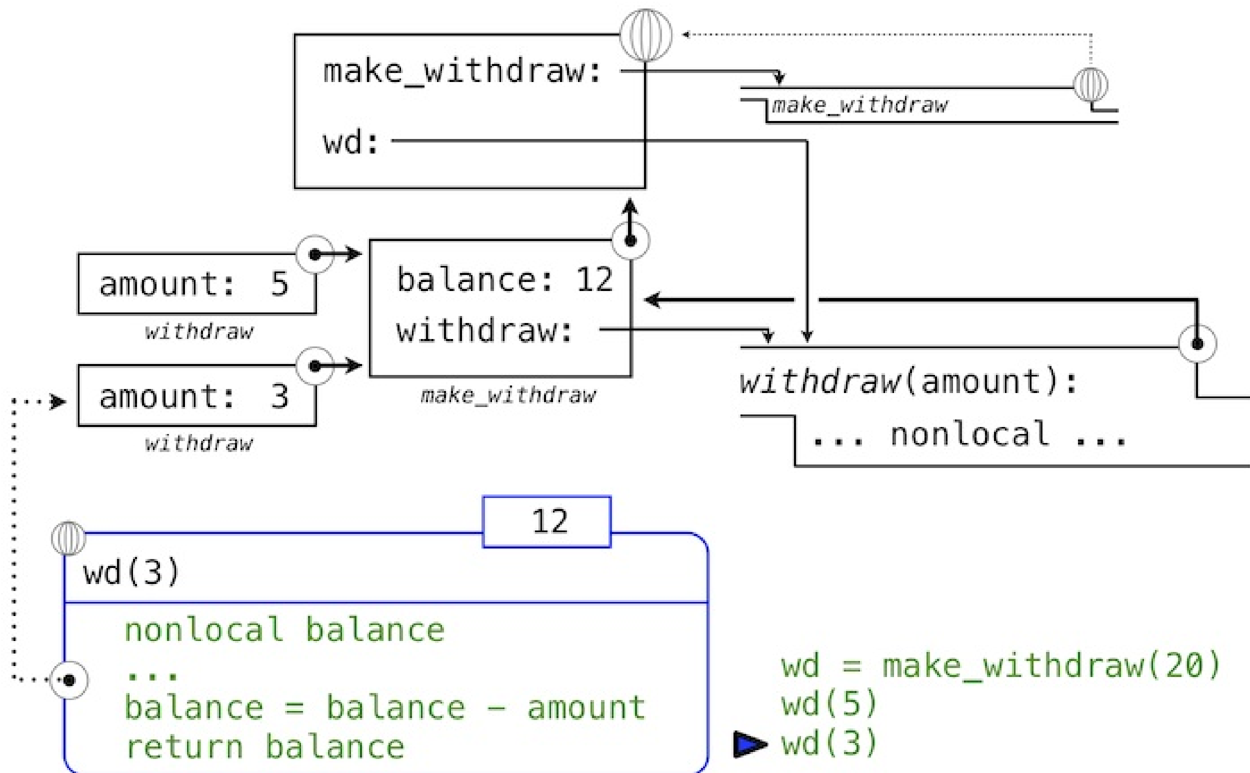
Our environment model of computation cleanly extends to explain the effects of non-local assignment. However, non-local assignment introduces some important nuances in the way we think about names and values.

Previously, our values did not change; only our names and bindings changed. When two names `a` and `b` were both bound to the value `4`, it did not matter whether they were bound to the same `4` or different `4`'s. As far as we could tell, there was only one `4` object that never changed.

However, functions with state do not behave this way. When two names `wd` and `wd2` are both bound to a *withdraw* function, it *does* matter whether they are bound to the same function or different instances of that function. Consider the following example, which contrasts the one we just analyzed.

```
>>> wd = make_withdraw(12)
>>> wd2 = wd
>>> wd2(1)
11
>>> wd(1)
10
```

In this case, calling the function named by `wd2` did change the value of the function named by `wd`, because both names refer to the same function. The environment diagram after these statements are executed shows this fact.



It is not unusual for two names to co-refer to the same value in the world, and so it is in our programs. But, as values change over time, we must be very careful to understand the effect of a change on other names that might refer to those values.

The key to correctly analyzing code with non-local assignment is to remember that only function calls can introduce new frames. Assignment statements always change bindings in existing frames. In this case, unless *make_withdraw* is called twice, there can be only one binding for `balance`.

Sameness and change. These subtleties arise because, by introducing non-pure functions that change the non-local environment, we have changed the nature of expressions. An expression that contains only pure function calls is *referentially transparent*; its value does not change if we substitute one of its subexpression with the value of that subexpression.

Re-binding operations violate the conditions of referential transparency because they do more than return a value; they change the environment. When we introduce arbitrary re-binding, we encounter a thorny epistemological issue: what it means for two values to be the same. In our environment model of computation, two separately defined functions are not the same, because changes to one may not be reflected in the other.

In general, so long as we never modify data objects, we can regard a compound data object to be precisely the totality of its pieces. For example, a rational number is determined by giving its numerator and its denominator. But this view is no longer valid in the presence of change, where a compound data object has an "identity" that is something different from the pieces of which it is composed. A bank account is still "the same" bank account even if we change the balance by making a withdrawal; conversely, we could have two bank accounts that happen to have the same balance, but are different objects.

Despite the complications it introduces, non-local assignment is a powerful tool for creating modular programs. Different parts of a program, which correspond to different environment frames, can evolve separately throughout program execution. Moreover, using functions with local state, we are able to implement mutable data types. In the remainder of this section, we introduce some of the most useful built-in data types in Python, along with methods for implementing those data types using functions with non-local assignment.

2.4.4 Lists

The `list` is Python's most useful and flexible sequence type. A list is similar to a tuple, but it is mutable. Method calls and assignment statements can change the contents of a list.

We can introduce many list editing operations through an example that illustrates the history of playing cards (drastically simplified). Comments in the examples describe the effect of each method invocation.

Playing cards were invented in China, perhaps around the 9th century. An early deck had three suits, which corresponded to denominations of money.

```
>>> chinese_suits = ['coin', 'string', 'myriad']  # A list literal
>>> suits = chinese_suits                         # Two names refer to the same list
```

As cards migrated to Europe (perhaps through Egypt), only the suit of coins remained in Spanish decks (*oro*).

```
>>> suits.pop()             # Removes and returns the final element
'myriad'
>>> suits.remove('string')  # Removes the first element that equals the argument
```

Three more suits were added (they evolved in name and design over time),

```
>>> suits.append('cup')              # Add an element to the end
>>> suits.extend(['sword', 'club'])  # Add all elements of a list to the end
```

and Italians called swords *spades*.

```
>>> suits[2] = 'spade'  # Replace an element
```

giving the suits of a traditional Italian deck of cards.

```
>>> suits
['coin', 'cup', 'spade', 'club']
```

The French variant that we use today in the U.S. changes the first two:

```
>>> suits[0:2] = ['heart', 'diamond']  # Replace a slice
>>> suits
['heart', 'diamond', 'spade', 'club']
```

Methods also exist for inserting, sorting, and reversing lists. All of these *mutation operations* change the value of the list; they do not create new list objects.

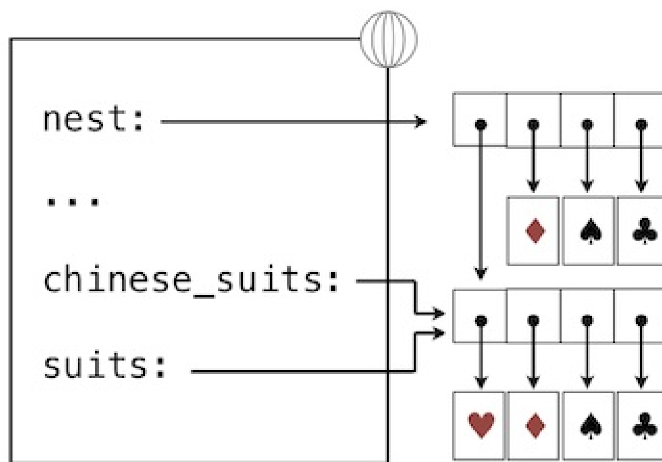
Sharing and Identity. Because we have been changing a single list rather than creating new lists, the object bound to the name `chinese_suits` has also changed, because it is the same list object that was bound to `suits`.

```
>>> chinese_suits  # This name co-refers with "suits" to the same list
['heart', 'diamond', 'spade', 'club']
```

Lists can be copied using the `list` constructor function. Changes to one list do not affect another, unless they share structure.

```
>>> nest = list(suits)  # Bind "nest" to a second list with the same elements
>>> nest[0] = suits     # Create a nested list
```

After this final assignment, we are left with the following environment, where lists are represented using box-and-pointer notation.



According to this environment, changing the list referenced by `suits` will affect the nested list that is the first element of `nest`, but not the other elements.

```
>>> suits.insert(2, 'Joker')  # Insert an element at index 2, shifting the rest
>>> nest
[['heart', 'diamond', 'Joker', 'spade', 'club'], 'diamond', 'spade', 'club']
```

And likewise, undoing this change in the first element of `nest` will change `suit` as well.

```
>>> nest[0].pop(2)
'Joker'
>>> suits
['heart', 'diamond', 'spade', 'club']
```

As a result of this last invocation of the `pop` method, we return to the environment depicted above.

Because two lists may have the same contents but in fact be different lists, we require a means to test whether two objects are the same. Python includes two comparison operators, called `is` and `is not`, that test whether two expressions in fact evaluate to the identical object. Two objects are identical if they are equal in their current value, and any change to one will always be reflected in the other. Identity is a stronger condition than equality.

```
>>> suits is nest[0]
True
>>> suits is ['heart', 'diamond', 'spade', 'club']
False
>>> suits == ['heart', 'diamond', 'spade', 'club']
True
```

The final two comparisons illustrate the difference between `is` and `==`. The former checks for identity, while the latter checks for the equality of contents.

List comprehensions. A list comprehension uses an extended syntax for creating lists, analogous to the syntax of generator expressions.

For example, the `unicodedata` module tracks the official names of every character in the Unicode alphabet. We can look up the characters corresponding to names, including those for card suits.

```
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]
['♡', '♢', '♤', '♧']
```

List comprehensions reinforce the paradigm of data processing using the conventional interface of sequences, as `list` is a sequence data type.

Further reading. Dive Into Python 3 has a chapter on comprehensions that includes examples of how to navigate a computer's file system using Python. The chapter introduces the `os` module, which for instance can list the contents of directories. This material is not part of the course, but recommended for anyone who wants to increase his or her Python expertise.

Implementation. Lists are sequences, like tuples. The Python language does not give us access to the implementation of lists, only to the sequence abstraction and the mutation methods we have introduced in this section. To overcome this language-enforced abstraction barrier, we can develop a functional implementation of lists, again using a recursive representation. This section also has a second purpose: to further our understanding of dispatch functions.

We will implement a list as a function that has a recursive list as its local state. Lists need to have an identity, like any mutable value. In particular, we cannot use `None` to represent an empty mutable list, because two empty lists are not identical values (e.g., appending to one does not append to the other), but `None is None`. On the other hand, two different functions that each have `empty_rlist` as their local state will suffice to distinguish two empty lists.

Our mutable list is a dispatch function, just as our functional implementation of a pair was a dispatch function. It checks the input "message" against known messages and takes an appropriate action for each different input. Our mutable list responds to five different messages. The first two implement the behaviors of the sequence abstraction. The next two add or remove the first element of the list. The final message returns a string representation of the whole list contents.

```
>>> def make_mutable_rlist():
        """Return a functional implementation of a mutable recursive list."""
        contents = empty_rlist
        def dispatch(message, value=None):
            nonlocal contents
            if message == 'len':
                return len_rlist(contents)
            elif message == 'getitem':
                return getitem_rlist(contents, value)
            elif message == 'push_first':
                contents = make_rlist(value, contents)
            elif message == 'pop_first':
                f = first(contents)
                contents = rest(contents)
                return f
            elif message == 'str':
                return str(contents)
        return dispatch
```

We can also add a convenience function to construct a functionally implemented recursive list from any built-in sequence, simply by adding each element in reverse order.

```
>>> def to_mutable_rlist(source):
        """Return a functional list with the same contents as source."""
        s = make_mutable_rlist()
        for element in reversed(source):
            s('push_first', element)
        return s
```

In the definition above, the function `reversed` takes and returns an iterable value; it is another example of a function that uses the conventional interface of sequences.

At this point, we can construct a functionally implemented lists. Note that the list itself is a function.

```
>>> s = to_mutable_rlist(suits)
>>> type(s)
<class 'function'>
>>> s('str')
"('heart', ('diamond', ('spade', ('club', None))))"
```

In addition, we can pass messages to the list `s` that change its contents, for instance removing the first element.

```
>>> s('pop_first')
'heart'
>>> s('str')
"('diamond', ('spade', ('club', None)))"
```

In principle, the operations `push_first` and `pop_first` suffice to make arbitrary changes to a list. We can always empty out the list entirely and then replace its old contents with the desired result.

Message passing. Given some time, we could implement the many useful mutation operations of Python lists, such as `extend` and `insert`. We would have a choice: we could implement them all as functions, which use the existing messages `pop_first` and `push_first` to make all changes. Alternatively, we could add additional `elif` clauses to the body of `dispatch`, each checking for a message (e.g., `'extend'`) and applying the appropriate change to `contents` directly.

This second approach, which encapsulates the logic for all operations on a data value within one function that responds to different messages, is called message passing. A program that uses message passing defines dispatch functions, each of which may have local state, and organizes computation by passing "messages" as the first argument to those functions. The messages are strings that correspond to particular behaviors.

One could imagine that enumerating all of these messages by name in the body of `dispatch` would become tedious and prone to error. Python dictionaries, introduced in the next section, provide a data type that will help us manage the mapping between messages and operations.

2.4.5 Dictionaries

Dictionaries are Python's built-in data type for storing and manipulating correspondence relationships. A dictionary contains key-value pairs, where both the keys and values are objects. The purpose of a dictionary is to provide an abstraction for storing and retrieving values that are indexed not by consecutive integers, but by descriptive keys.

Strings commonly serve as keys, because strings are our conventional representation for names of things. This dictionary literal gives the values of various Roman numerals.

```
>>> numerals = {'I': 1.0, 'V': 5, 'X': 10}
```

Looking up values by their keys uses the element selection operator that we previously applied to sequences.

```
>>> numerals['X']
10
```

A dictionary can have at most one value for each key. Adding new key-value pairs and changing the existing value for a key can both be achieved with assignment statements.

```
>>> numerals['I'] = 1
>>> numerals['L'] = 50
>>> numerals
{'I': 1, 'X': 10, 'L': 50, 'V': 5}
```

Notice that `'L'` was not added to the end of the output above. Dictionaries are unordered collections of key-value pairs. When we print a dictionary, the keys and values are rendered in some order, but as users of the language we cannot predict what that order will be.

The dictionary abstraction also supports various methods of iterating of the contents of the dictionary as a whole. The methods `keys`, `values`, and `items` all return iterable values.

```
>>> sum(numerals.values())
66
```

A list of key-value pairs can be converted into a dictionary by calling the `dict` constructor function.

```
>>> dict([(3, 9), (4, 16), (5, 25)])
{3: 9, 4: 16, 5: 25}
```

Dictionaries do have some restrictions:

- A key of a dictionary cannot be an object of a mutable built-in type.
- There can be at most one value for a given key.

This first restriction is tied to the underlying implementation of dictionaries in Python. The details of this implementation are not a topic of this course. Intuitively, consider that the key tells Python where to find that key-value pair in memory; if the key changes, the location of the pair may be lost.

The second restriction is a consequence of the dictionary abstraction, which is designed to store and retrieve values for keys. We can only retrieve *the* value for a key if at most one such value exists in the dictionary.

A useful method implemented by dictionaries is `get`, which returns either the value for a key, if the key is present, or a default value. The arguments to `get` are the key and the default value.

```
>>> numerals.get('A', 0)
0
>>> numerals.get('V', 0)
5
```

Dictionaries also have a comprehension syntax analogous to those of lists and generator expressions. Evaluating a dictionary comprehension yields a new dictionary object.

```
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
```

Implementation. We can implement an abstract data type that conforms to the dictionary abstraction as a list of records, each of which is a two-element list consisting of a key and the associated value.

```
>>> def make_dict():
        """Return a functional implementation of a dictionary."""
        records = []
        def getitem(key):
            for k, v in records:
                if k == key:
                    return v
        def setitem(key, value):
            for item in records:
                if item[0] == key:
                    item[1] = value
                    return
            records.append([key, value])
        def dispatch(message, key=None, value=None):
            if message == 'getitem':
                return getitem(key)
            elif message == 'setitem':
                setitem(key, value)
            elif message == 'keys':
                return tuple(k for k, _ in records)
            elif message == 'values':
                return tuple(v for _, v in records)
        return dispatch
```

Again, we use the message passing method to organize our implementation. We have supported four messages: `getitem`, `setitem`, `keys`, and `values`. To look up a value for a key, we iterate through the records to find a matching key. To insert a value for a key, we iterate through the records to see if there is already a record with that key. If not, we form a new record. If there already is a record with this key, we set the value of the record to the designated new value.

We can now use our implementation to store and retrieve values.

```
>>> d = make_dict()
>>> d('setitem', 3, 9)
>>> d('setitem', 4, 16)
>>> d('getitem', 3)
9
>>> d('getitem', 4)
16
>>> d('keys')
(3, 4)
>>> d('values')
(9, 16)
```

This implementation of a dictionary is *not* optimized for fast record lookup, because each response to the message `'getitem'` must iterate through the entire list of `records`. The built-in dictionary type is considerably more efficient.

2.4.6 Example: Propagating Constraints

Mutable data allows us to simulate systems with change, but also allows us to build new kinds of abstractions. In this extended example, we combine nonlocal assignment, lists, and dictionaries to build a *constraint-based system* that supports computation in multiple directions. Expressing programs as constraints is a type of *declarative programming*, in which a programmer declares the structure of a problem to be solved, but abstracts away the details of exactly how the solution to the problem is computed.

Computer programs are traditionally organized as one-directional computations, which perform operations on pre-specified arguments to produce desired outputs. On the other hand, we often want to model systems in terms of relations among quantities. For example, we previously considered the ideal gas law, which relates the pressure (`p`), volume (`v`), quantity (`n`), and temperature (`t`) of an ideal gas via Boltzmann's constant (`k`):

```
p * v = n * k * t
```

Such an equation is not one-directional. Given any four of the quantities, we can use this equation to compute the fifth. Yet translating the equation into a traditional computer language would force us to choose one of the quantities to be computed in terms of the other four. Thus, a function for computing the pressure could not be used to compute the temperature, even though the computations of both quantities arise from the same equation.

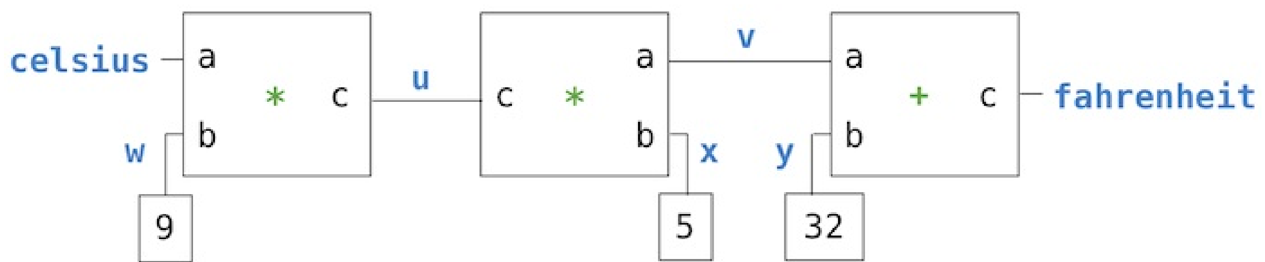
In this section, we sketch the design of a general model of linear relationships. We define primitive constraints that hold between quantities, such as an `adder(a, b, c)` constraint that enforces the mathematical relationship `a + b = c` .

We also define a means of combination, so that primitive constraints can be combined to express more complex relations. In this way, our program resembles a programming language. We combine constraints by constructing a network in which constraints are joined by connectors. A connector is an object that "holds" a value and may participate in one or more constraints.

For example, we know that the relationship between Fahrenheit and Celsius temperatures is:

```
9 * c = 5 * (f - 32)
```

This equation is a complex constraint between `c` and `f` . Such a constraint can be thought of as a network consisting of primitive `adder` , `multiplier` , and `constant` constraints.



In this figure, we see on the left a multiplier box with three terminals, labeled `a`, `b`, and `c`. These connect the multiplier to the rest of the network as follows: The `a` terminal is linked to a connector `celsius`, which will hold the Celsius temperature. The `b` terminal is linked to a connector `w`, which is also linked to a constant box that holds `9`. The `c` terminal, which the multiplier box constrains to be the product of `a` and `b`, is linked to the `c` terminal of another multiplier box, whose `b` is connected to a constant `5` and whose `a` is connected to one of the terms in the sum constraint.

Computation by such a network proceeds as follows: When a connector is given a value (by the user or by a constraint box to which it is linked), it awakens all of its associated constraints (except for the constraint that just awakened it) to inform them that it has a value. Each awakened constraint box then polls its connectors to see if there is enough information to determine a value for a connector. If so, the box sets that connector, which then awakens all of its associated constraints, and so on. For instance, in conversion between Celsius and Fahrenheit, `w`, `x`, and `y` are immediately set by the constant boxes to `9`, `5`, and `32`, respectively. The connectors awaken the multipliers and the adder, which determine that there is not enough information to proceed. If the user (or some other part of the network) sets the `celsius` connector to a value (say `25`), the leftmost multiplier will be awakened, and it will set `u` to `25 * 9 = 225`. Then `u` awakens the second multiplier, which sets `v` to `45`, and `v` awakens the adder, which sets the `fahrenheit` connector to `77`.

Using the Constraint System. To use the constraint system to carry out the temperature computation outlined above, we first create two named connectors, `celsius` and `fahrenheit`, by calling the `make_connector` constructor.

```
>>> celsius = make_connector('Celsius')
>>> fahrenheit = make_connector('Fahrenheit')
```

Then, we link these connectors into a network that mirrors the figure above. The function `make_converter` assembles the various connectors and constraints in the network.

```
>>> def make_converter(c, f):
        """Connect c to f with constraints to convert from Celsius to Fahrenheit."""
        u, v, w, x, y = [make_connector() for _ in range(5)]
        multiplier(c, w, u)
        multiplier(v, x, u)
        adder(v, y, f)
        constant(w, 9)
        constant(x, 5)
        constant(y, 32)
```

```
>>> make_converter(celsius, fahrenheit)
```

We will use a message passing system to coordinate constraints and connectors. Instead of using functions to answer messages, we will use dictionaries. A dispatch dictionary will have string-valued keys that denote the messages it accepts. The values associated with those keys will be the responses to those messages.

Constraints are dictionaries that do not hold local states themselves. Their responses to messages are non-pure functions that change the connectors that they constrain.

Connectors are dictionaries that hold a current value and respond to messages that manipulate that value. Constraints will not change the value of connectors directly, but instead will do so by sending messages, so that the connector can notify other constraints in response to the change. In this way, a connector represents a number, but also encapsulates connector behavior.

One message we can send to a connector is to set its value. Here, we (the `'user'`) set the value of `celsius` to `25`.

```
>>> celsius['set_val']('user', 25)
Celsius = 25
Fahrenheit = 77.0
```

Not only does the value of `celsius` change to `25`, but its value propagates through the network, and so the value of `fahrenheit` is changed as well. These changes are printed because we named these two connectors when we constructed them.

Now we can try to set `fahrenheit` to a new value, say `212`.

```
>>> fahrenheit['set_val']('user', 212)
Contradiction detected: 77.0 vs 212
```

The connector complains that it has sensed a contradiction: Its value is `77.0`, and someone is trying to set it to `212`. If we really want to reuse the network with new values, we can tell `celsius` to forget its old value:

```
>>> celsius['forget']('user')
Celsius is forgotten
Fahrenheit is forgotten
```

The connector `celsius` finds that the `user`, who set its value originally, is now retracting that value, so `celsius` agrees to lose its value, and it informs the rest of the network of this fact. This information eventually propagates to `fahrenheit`, which now finds that it has no reason for continuing to believe that its own value is `77`. Thus, it also gives up its value.

Now that `fahrenheit` has no value, we are free to set it to `212`:

```
>>> fahrenheit['set_val']('user', 212)
Fahrenheit = 212
Celsius = 100.0
```

This new value, when propagated through the network, forces `celsius` to have a value of `100`. We have used the very same network to compute `celsius` given `fahrenheit` and to compute `fahrenheit` given `celsius`. This non-directionality of computation is the distinguishing feature of constraint-based systems.

Implementing the Constraint System. As we have seen, connectors are dictionaries that map message names to function and data values. We will implement connectors that respond to the following messages:

- `connector['set_val'](source, value)` indicates that the `source` is requesting the connector to set its current value to `value`.
- `connector['has_val']()` returns whether the connector already has a value.
- `connector['val']` is the current value of the connector.
- `connector['forget'](source)` tells the connector that the `source` is requesting it to forget its value.
- `connector['connect'](source)` tells the connector to participate in a new constraint, the `source`.

Constraints are also dictionaries, which receive information from connectors by means of two messages:

- `constraint['new_val']()` indicates that some connector that is connected to the constraint has a new value.
- `constraint['forget']()` indicates that some connector that is connected to the constraint has forgotten its value.

When constraints receive these messages, they propagate them appropriately to other connectors.

The `adder` function constructs an adder constraint over three connectors, where the first two must add to the third: `a + b = c`. To support multidirectional constraint propagation, the adder must also specify that it subtracts `a` from `c` to get `b` and likewise subtracts `b` from `c` to get `a`.

```
>>> from operator import add, sub
>>> def adder(a, b, c):
        """The constraint that a + b = c."""
        return make_ternary_constraint(a, b, c, add, sub, sub)
```

We would like to implement a generic ternary (three-way) constraint, which uses the three connectors and three functions from `adder` to create a constraint that accepts `new_val` and `forget` messages. The response to messages are local functions, which are placed in a dictionary called `constraint`.

```
>>> def make_ternary_constraint(a, b, c, ab, ca, cb):
        """The constraint that ab(a,b)=c and ca(c,a)=b and cb(c,b) = a."""
        def new_value():
            av, bv, cv = [connector['has_val']() for connector in (a, b, c)]
            if av and bv:
                c['set_val'](constraint, ab(a['val'], b['val']))
            elif av and cv:
                b['set_val'](constraint, ca(c['val'], a['val']))
            elif bv and cv:
                a['set_val'](constraint, cb(c['val'], b['val']))
        def forget_value():
            for connector in (a, b, c):
                connector['forget'](constraint)
        constraint = {'new_val': new_value, 'forget': forget_value}
        for connector in (a, b, c):
            connector['connect'](constraint)
        return constraint
```

The dictionary called `constraint` is a dispatch dictionary, but also the constraint object itself. It responds to the two messages that constraints receive, but is also passed as the `source` argument in calls to its connectors.

The constraint's local function `new_value` is called whenever the constraint is informed that one of its connectors has a value. This function first checks to see if both `a` and `b` have values. If so, it tells `c` to set its value to the return value of function `ab`, which is `add` in the case of an `adder`. The constraint passes *itself* (`constraint`) as the `source` argument of the connector, which is the adder object. If `a` and `b` do not both have values, then the constraint checks `a` and `c`, and so on.

If the constraint is informed that one of its connectors has forgotten its value, it requests that all of its connectors now forget their values. (Only those values that were set by this constraint are actually lost.)

A `multiplier` is very similar to an `adder`.

```
>>> from operator import mul, truediv
>>> def multiplier(a, b, c):
        """The constraint that a * b = c."""
        return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```

A constant is a constraint as well, but one that is never sent any messages, because it involves only a single connector that it sets on construction.

```
>>> def constant(connector, value):
        """The constraint that connector = value."""
        constraint = {}
        connector['set_val'](constraint, value)
        return constraint
```

These three constraints are sufficient to implement our temperature conversion network.

Representing connectors. A connector is represented as a dictionary that contains a value, but also has response functions with local state. The connector must track the `informant` that gave it its current value, and a list of `constraints` in which it participates.

The constructor `make_connector` has local functions for setting and forgetting values, which are the responses to messages from constraints.

```
>>> def make_connector(name=None):
        """A connector between constraints."""
        informant = None
        constraints = []
        def set_value(source, value):
            nonlocal informant
            val = connector['val']
            if val is None:
                informant, connector['val'] = source, value
                if name is not None:
                    print(name, '=', value)
                inform_all_except(source, 'new_val', constraints)
            else:
                if val != value:
                    print('Contradiction detected:', val, 'vs', value)
        def forget_value(source):
            nonlocal informant
            if informant == source:
                informant, connector['val'] = None, None
                if name is not None:
                    print(name, 'is forgotten')
                inform_all_except(source, 'forget', constraints)
        connector = {'val': None,
                     'set_val': set_value,
                     'forget': forget_value,
                     'has_val': lambda: connector['val'] is not None,
                     'connect': lambda source: constraints.append(source)}
        return connector
```

A connector is again a dispatch dictionary for the five messages used by constraints to communicate with connectors. Four responses are functions, and the final response is the value itself.

The local function `set_value` is called when there is a request to set the connector's value. If the connector does not currently have a value, it will set its value and remember as `informant` the source constraint that requested the value to be set. Then the connector will notify all of its participating constraints except the constraint that requested the value to be set. This is accomplished using the following iterative function.

```
>>> def inform_all_except(source, message, constraints):
        """Inform all constraints of the message, except source."""
        for c in constraints:
            if c != source:
                c[message]()
```

If a connector is asked to forget its value, it calls the local function `forget-value`, which first checks to make sure that the request is coming from the same constraint that set the value originally. If so, the connector informs its associated constraints about the loss of the value.

The response to the message `has_val` indicates whether the connector has a value. The response to the message `connect` adds the source constraint to the list of constraints.

The constraint program we have designed introduces many ideas that will appear again in object-oriented programming. Constraints and connectors are both abstractions that are manipulated through messages. When the value of a connector is changed, it is changed via a message that not only changes the value, but validates it (checking the source) and propagates its effects (informing other constraints). In fact, we will use a similar architecture of dictionaries with string-valued keys and functional values to implement an object-oriented system later in this chapter.

2.5 Object-Oriented Programming

Object-oriented programming (OOP) is a method for organizing programs that brings together many of the ideas introduced in this chapter. Like abstract data types, objects create an abstraction barrier between the use and implementation of data. Like dispatch dictionaries in message passing, objects respond to behavioral requests. Like mutable data structures, objects have local state that is not directly accessible from the global environment. The Python object system provides new syntax to ease the task of implementing all of these useful techniques for organizing programs.

But the object system offers more than just convenience; it enables a new metaphor for designing programs in which several independent agents interact within the computer. Each object bundles together local state and behavior in a way that hides the complexity of both behind a data abstraction. Our example of a constraint program began to develop this metaphor by passing messages between constraints and connectors. The Python object system extends this metaphor with new ways to express how different parts of a program relate to and communicate with each other. Not only do objects pass messages, they also share behavior among other objects of the same type and inherit characteristics from related types.

The paradigm of object-oriented programming has its own vocabulary that reinforces the object metaphor. We have seen that an object is a data value that has methods and attributes, accessible via dot notation. Every object also has a type, called a *class*. New classes can be defined in Python, just as new functions can be defined.

2.5.1 Objects and Classes

A class serves as a template for all objects whose type is that class. Every object is an instance of some particular class. The objects we have used so far all have built-in classes, but new classes can be defined similarly to how new functions can be defined. A class definition specifies the attributes and methods shared among objects of that class. We will introduce the class statement by revisiting the example of a bank account.

When introducing local state, we saw that bank accounts are naturally modeled as mutable values that have a `balance`. A bank account object should have a `withdraw` method that updates the account balance and returns the requested amount, if it is available. We would like additional behavior to complete the account abstraction: a bank account should be able to return its current balance, return the name of the account holder, and accept deposits.

An `Account` class allows us to create multiple instances of bank accounts. The act of creating a new object instance is known as *instantiating* the class. The syntax in Python for instantiating a class is identical to the syntax of calling a function. In this case, we call `Account` with the argument `'Jim'`, the account holder's name.

```
>>> a = Account('Jim')
```

An *attribute* of an object is a name-value pair associated with the object, which is accessible via dot notation. The attributes specific to a particular object, as opposed to all objects of a class, are called *instance attributes*. Each `Account` has its own balance and account holder name, which are examples of instance attributes. In the broader programming community, instance attributes may also be called *fields*, *properties*, or *instance variables*.

```
>>> a.holder
'Jim'
>>> a.balance
0
```

Functions that operate on the object or perform object-specific computations are called methods. The side effects and return value of a method can depend upon, and change, other attributes of the object. For example, `deposit` is a method of our `Account` object `a`. It takes one argument, the amount to deposit, changes the `balance` attribute of the object, and returns the resulting balance.

```
>>> a.deposit(15)
15
```

In OOP, we say that methods are *invoked* on a particular object. As a result of invoking the `withdraw` method, either the withdrawal is approved and the amount is deducted and returned, or the request is declined and the account prints an error message.

```
>>> a.withdraw(10)  # The withdraw method returns the balance after withdrawal
5
>>> a.balance       # The balance attribute has changed
5
>>> a.withdraw(10)
'Insufficient funds'
```

As illustrated above, the behavior of a method can depend upon the changing attributes of the object. Two calls to `withdraw` with the same argument return different results.

2.5.2 Defining Classes

User-defined classes are created by `class` statements, which consist of a single clause. A class statement defines the class name and a base class (discussed in the section on Inheritance), then includes a suite of statements to define the attributes of the class:

```
class <name>(<base class>):
    <suite>
```

When a class statement is executed, a new class is created and bound to `<name>` in the first frame of the current environment. The suite is then executed. Any names bound within the `<suite>` of a `class` statement, through `def` or assignment statements, create or modify attributes of the class.

Classes are typically organized around manipulating instance attributes, which are the name-value pairs associated not with the class itself, but with each object of that class. The class specifies the instance attributes of its objects by defining a method for initializing new objects. For instance, part of initializing an object of the `Account` class is to assign it a starting balance of 0.

The `<suite>` of a `class` statement contains `def` statements that define new methods for objects of that class. The method that initializes objects has a special name in Python, `__init__` (two underscores on each side of "init"), and is called the *constructor* for the class.

```
>>> class Account(object):
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
```

The `__init__` method for `Account` has two formal parameters. The first one, `self`, is bound to the newly created `Account` object. The second parameter, `account_holder`, is bound to the argument passed to the class when it is called to be instantiated.

The constructor binds the instance attribute name `balance` to `0`. It also binds the attribute name `holder` to the value of the name `account_holder`. The formal parameter `account_holder` is a local name to the `__init__` method. On the other hand, the name `holder` that is bound via the final assignment statement persists, because it is stored as an attribute of `self` using dot notation.

Having defined the `Account` class, we can instantiate it.

```
>>> a = Account('Jim')
```

This "call" to the `Account` class creates a new object that is an instance of `Account`, then calls the constructor function `__init__` with two arguments: the newly created object and the string `'Jim'`. By convention, we use the parameter name `self` for the first argument of a constructor, because it is bound to the object being instantiated. This convention is adopted in virtually all Python code.

Now, we can access the object's `balance` and `holder` using dot notation.

```
>>> a.balance
0
>>> a.holder
'Jim'
```

Identity. Each new account instance has its own balance attribute, the value of which is independent of other objects of the same class.

```
>>> b = Account('Jack')
>>> b.balance = 200
>>> [acc.balance for acc in (a, b)]
[0, 200]
```

To enforce this separation, every object that is an instance of a user-defined class has a unique identity. Object identity is compared using the `is` and `is not` operators.

```
>>> a is a
True
>>> a is not b
True
```

Despite being constructed from identical calls, the objects bound to `a` and `b` are not the same. As usual, binding an object to a new name using assignment does not create a new object.

```
>>> c = a
>>> c is a
True
```

New objects that have user-defined classes are only created when a class (such as `Account`) is instantiated with call expression syntax.

Methods. Object methods are also defined by a `def` statement in the suite of a `class` statement. Below, `deposit` and `withdraw` are both defined as methods on objects of the `Account` class.

```
>>> class Account(object):
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

While method definitions do not differ from function definitions in how they are declared, method definitions do have a different effect. The function value that is created by a `def` statement within a `class` statement is bound to the declared name, but bound locally within the class as an attribute. That value is invoked as a method using dot notation from an instance of the class.

Each method definition again includes a special first parameter `self`, which is bound to the object on which the method is invoked. For example, let us say that `deposit` is invoked on a particular `Account` object and passed a single argument value: the amount deposited. The object itself is bound to `self`, while the argument is bound to `amount`. All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

To invoke these methods, we again use dot notation, as illustrated below.

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
>>> tom_account.withdraw(90)
10
>>> tom_account.withdraw(90)
'Insufficient funds'
>>> tom_account.holder
'Tom'
```

When a method is invoked via dot notation, the object itself (bound to `tom_account`, in this case) plays a dual role. First, it determines what the name `withdraw` means; `withdraw` is not a name in the environment, but instead a name that is local to the `Account` class. Second, it is bound to the first parameter `self` when the `withdraw` method is invoked. The details of the procedure for evaluating dot notation follow in the next section.

2.5.3 Message Passing and Dot Expressions

Methods, which are defined in classes, and instance attributes, which are typically assigned in constructors, are the fundamental elements of object-oriented programming. These two concepts replicate much of the behavior of a dispatch dictionary in a message passing

implementation of a data value. Objects take messages using dot notation, but instead of those messages being arbitrary string-valued keys, they are names local to a class. Objects also have named local state values (the instance attributes), but that state can be accessed and manipulated using dot notation, without having to employ `nonlocal` statements in the implementation.

The central idea in message passing was that data values should have behavior by responding to messages that are relevant to the abstract type they represent. Dot notation is a syntactic feature of Python that formalizes the message passing metaphor. The advantage of using a language with a built-in object system is that message passing can interact seamlessly with other language features, such as assignment statements. We do not require different messages to "get" or "set" the value associated with a local attribute name; the language syntax allows us to use the message name directly.

Dot expressions. The code fragment `tom_account.deposit` is called a *dot expression*. A dot expression consists of an expression, a dot, and a name:

```
<expression> . <name>
```

The `<expression>` can be any valid Python expression, but the `<name>` must be a simple name (not an expression that evaluates to a name). A dot expression evaluates to the value of the attribute with the given `<name>`, for the object that is the value of the `<expression>`.

The built-in function `getattr` also returns an attribute for an object by name. It is the function equivalent of dot notation. Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch dictionary.

```
>>> getattr(tom_account, 'balance')
10
```

We can also test whether an object has a named attribute with `hasattr`.

```
>>> hasattr(tom_account, 'deposit')
True
```

The attributes of an object include all of its instance attributes, along with all of the attributes (including methods) defined in its class. Methods are attributes of the class that require special handling.

Method and functions. When a method is invoked on an object, that object is implicitly passed as the first argument to the method. That is, the object that is the value of the `<expression>` to the left of the dot is passed automatically as the first argument to the

method named on the right side of the dot expression. As a result, the object is bound to the parameter `self`.

To achieve automatic `self` binding, Python distinguishes between *functions*, which we have been creating since the beginning of the course, and *bound methods*, which couple together a function and the object on which that method will be invoked. A bound method value is already associated with its first argument, the instance on which it was invoked, which will be named `self` when the method is called.

We can see the difference in the interactive interpreter by calling `type` on the returned values of dot expressions. As an attribute of a class, a method is just a function, but as an attribute of an instance, it is a bound method:

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```

These two results differ only in the fact that the first is a standard two-argument function with parameters `self` and `amount`. The second is a one-argument method, where the name `self` will be bound to the object named `tom_account` automatically when the method is called, while the parameter `amount` will be bound to the argument passed to the method. Both of these values, whether function values or bound method values, are associated with the same `deposit` function body.

We can call `deposit` in two ways: as a function and as a bound method. In the former case, we must supply an argument for the `self` parameter explicitly. In the latter case, the `self` parameter is bound automatically.

```
>>> Account.deposit(tom_account, 1001)  # The deposit function requires 2 arguments
1011
>>> tom_account.deposit(1000)           # The deposit method takes 1 argument
2011
```

The function `getattr` behaves exactly like dot notation: if its first argument is an object but the name is a method defined in the class, then `getattr` returns a bound method value. On the other hand, if the first argument is a class, then `getattr` returns the attribute value directly, which is a plain function.

Practical guidance: naming conventions. Class names are conventionally written using the CapWords convention (also called CamelCase because the capital letters in the middle of a name are like humps). Method names follow the standard convention of naming functions using lowercased words separated by underscores.

In some cases, there are instance variables and methods that are related to the maintenance and consistency of an object that we don't want users of the object to see or use. They are not part of the abstraction defined by a class, but instead part of the implementation. Python's convention dictates that if an attribute name starts with an underscore, it should only be accessed within methods of the class itself, rather than by users of the class.

2.5.4 Class Attributes

Some attribute values are shared across all objects of a given class. Such attributes are associated with the class itself, rather than any individual instance of the class. For instance, let us say that a bank pays interest on the balance of accounts at a fixed interest rate. That interest rate may change, but it is a single value shared across all accounts.

Class attributes are created by assignment statements in the suite of a `class` statement, outside of any method definition. In the broader developer community, class attributes may also be called class variables or static variables. The following class statement creates a class attribute for `Account` with the name `interest`.

```
>>> class Account(object):
        interest = 0.02            # A class attribute
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        # Additional methods would be defined here
```

This attribute can still be accessed from any instance of the class.

```
>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```

However, a single assignment statement to a class attribute changes the value of the attribute for all instances of the class.

```
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

Attribute names. We have introduced enough complexity into our object system that we have to specify how names are resolved to particular attributes. After all, we could easily have a class attribute and an instance attribute with the same name.

As we have seen, a dot expressions consist of an expression, a dot, and a name:

```
<expression> . <name>
```

To evaluate a dot expression:

1. Evaluate the `<expression>` to the left of the dot, which yields the *object* of the dot expression.
2. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If `<name>` does not appear among instance attributes, then `<name>` is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is returned instead.

In this evaluation procedure, instance attributes are found before class attributes, just as local names have priority over global in an environment. Methods defined within the class are bound to the object of the dot expression during the third step of this evaluation procedure. The procedure for looking up a name in a class has additional nuances that will arise shortly, once we introduce class inheritance.

Assignment. All assignment statements that contain a dot expression on their left-hand side affect attributes for the object of that dot expression. If the object is an instance, then assignment sets an instance attribute. If the object is a class, then assignment sets a class attribute. As a consequence of this rule, assignment to an attribute of an object cannot affect the attributes of its class. The examples below illustrate this distinction.

If we assign to the named attribute `interest` of an account instance, we create a new instance attribute that has the same name as the existing class attribute.

```
>>> jim_account.interest = 0.08
```

and that attribute value will be returned from a dot expression.

```
>>> jim_account.interest
0.08
```

However, the class attribute `interest` still retains its original value, which is returned for all other accounts.

```
>>> tom_account.interest
0.04
```

Changes to the class attribute `interest` will affect `tom_account`, but the instance attribute for `jim_account` will be unaffected.

```
>>> Account.interest = 0.05  # changing the class attribute
>>> tom_account.interest     # changes instances without like-named instance attributes
0.05
>>> jim_account.interest     # but the existing instance attribute is unaffected
0.08
```

2.5.5 Inheritance

When working in the OOP paradigm, we often find that different abstract data types are related. In particular, we find that similar classes differ in their amount of specialization. Two classes may have similar attributes, but one represents a special case of the other.

For example, we may want to implement a checking account, which is different from a standard account. A checking account charges an extra $1 for each withdrawal and has a lower interest rate. Here, we demonstrate the desired behavior.

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # withdrawals decrease balance by an extra charge
14
```

A `CheckingAccount` is a specialization of an `Account`. In OOP terminology, the generic account will serve as the base class of `CheckingAccount`, while `CheckingAccount` will be a subclass of `Account`. (The terms *parent class* and *superclass* are also used for the base class, while *child class* is also used for the subclass.)

A subclass *inherits* the attributes of its base class, but may *override* certain attributes, including certain methods. With inheritance, we only specify what is different between the subclass and the base class. Anything that we leave unspecified in the subclass is automatically assumed to behave just as it would for the base class.

Inheritance also has a role in our object metaphor, in addition to being a useful organizational feature. Inheritance is meant to represent *is-a* relationships between classes, which contrast with *has-a* relationships. A checking account *is-a* specific type of account, so having a `CheckingAccount` inherit from `Account` is an appropriate use of inheritance. On the other hand, a bank *has-a* list of bank accounts that it manages, so neither should inherit from the other. Instead, a list of account objects would be naturally expressed as an instance attribute of a bank object.

2.5.6 Using Inheritance

We specify inheritance by putting the base class in parentheses after the class name. First, we give a full implementation of the `Account` class, which includes docstrings for the class and its methods.

```
>>> class Account(object):
        """A bank account that has a non-negative balance."""
        interest = 0.02
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            """Increase the account balance by amount and return the new balance."""
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            """Decrease the account balance by amount and return the new balance."""
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

A full implementation of `CheckingAccount` appears below.

```
>>> class CheckingAccount(Account):
        """A bank account that charges for withdrawals."""
        withdraw_charge = 1
        interest = 0.01
        def withdraw(self, amount):
            return Account.withdraw(self, amount + self.withdraw_charge)
```

Here, we introduce a class attribute `withdraw_charge` that is specific to the `CheckingAccount` class. We assign a lower value to the `interest` attribute. We also define a new `withdraw` method to override the behavior defined in the `Account` class. With no further statements in the class suite, all other behavior is inherited from the base class `Account`.

```
>>> checking = CheckingAccount('Sam')
>>> checking.deposit(10)
10
>>> checking.withdraw(5)
4
>>> checking.interest
0.01
```

The expression `checking.deposit` evaluates to a bound method for making deposits, which was defined in the `Account` class. When Python resolves a name in a dot expression that is not an attribute of the instance, it looks up the name in the class. In fact, the act of "looking up" a name in a class tries to find that name in every base class in the inheritance chain for the original object's class. We can define this procedure recursively. To look up a name in a class.

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

In the case of `deposit`, Python would have looked for the name first on the instance, and then in the `CheckingAccount` class. Finally, it would look in the `Account` class, where `deposit` is defined. According to our evaluation rule for dot expressions, since `deposit` is a function looked up in the class for the `checking` instance, the dot expression evaluates to a bound method value. That method is invoked with the argument `10`, which calls the deposit method with `self` bound to the `checking` object and `amount` bound to `10`.

The class of an object stays constant throughout. Even though the `deposit` method was found in the `Account` class, `deposit` is called with `self` bound to an instance of `CheckingAccount`, not of `Account`.

Calling ancestors. Attributes that have been overridden are still accessible via class objects. For instance, we implemented the `withdraw` method of `CheckingAccount` by calling the `withdraw` method of `Account` with an argument that included the `withdraw_charge`.

Notice that we called `self.withdraw_charge` rather than the equivalent `CheckingAccount.withdraw_charge`. The benefit of the former over the latter is that a class that inherits from `CheckingAccount` might override the withdrawal charge. If that is the case, we would like our implementation of `withdraw` to find that new value instead of the old one.

2.5.7 Multiple Inheritance

Python supports the concept of a subclass inheriting attributes from multiple base classes, a language feature called *multiple inheritance*.

Suppose that we have a `SavingsAccount` that inherits from `Account`, but charges customers a small fee every time they make a deposit.

```
>>> class SavingsAccount(Account):
        deposit_charge = 2
        def deposit(self, amount):
            return Account.deposit(self, amount - self.deposit_charge)
```

Then, a clever executive conceives of an `AsSeenOnTVAccount` account with the best features of both `CheckingAccount` and `SavingsAccount`: withdrawal fees, deposit fees, and a low interest rate. It's both a checking and a savings account in one! "If we build it," the executive reasons, "someone will sign up and pay all those fees. We'll even give them a dollar."

```
>>> class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
        def __init__(self, account_holder):
            self.holder = account_holder
            self.balance = 1           # A free dollar!
```

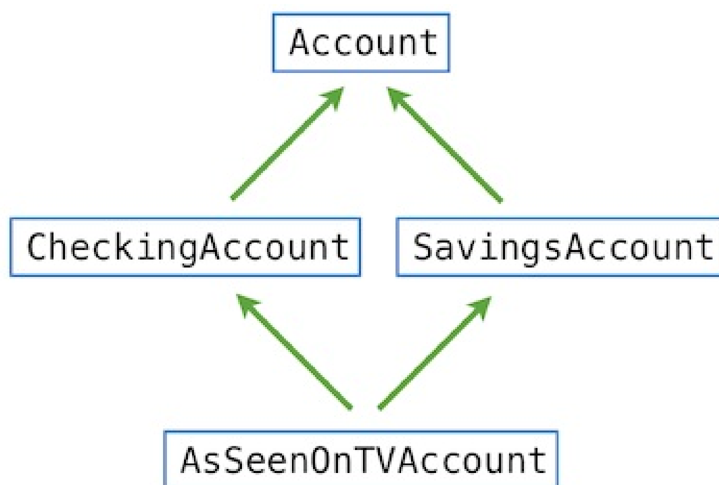
In fact, this implementation is complete. Both withdrawal and deposits will generate fees, using the function definitions in `CheckingAccount` and `SavingsAccount` respectively.

```
>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)            # $2 fee from SavingsAccount.deposit
19
>>> such_a_deal.withdraw(5)            # $1 fee from CheckingAccount.withdraw
13
```

Non-ambiguous references are resolved correctly as expected:

```
>>> such_a_deal.deposit_charge
2
>>> such_a_deal.withdraw_charge
1
```

But what about when the reference is ambiguous, such as the reference to the `withdraw` method that is defined in both `Account` and `CheckingAccount`? The figure below depicts an *inheritance graph* for the `AsSeenOnTVAccount` class. Each arrow points from a subclass to a base class.



For a simple "diamond" shape like this, Python resolves names from left to right, then upwards. In this example, Python checks for an attribute name in the following classes, in order, until an attribute with that name is found:

```
AsSeenOnTVAccount, CheckingAccount, SavingsAccount, Account, object
```

There is no correct solution to the inheritance ordering problem, as there are cases in which we might prefer to give precedence to certain inherited classes over others. However, any programming language that supports multiple inheritance must select some ordering in a consistent way, so that users of the language can predict the behavior of their programs.

Further reading. Python resolves this name using a recursive algorithm called the C3 Method Resolution Ordering. The method resolution order of any class can be queried using the `mro` method on all classes.

```
>>> [c.__name__ for c in AsSeenOnTVAccount.mro()]
['AsSeenOnTVAccount', 'CheckingAccount', 'SavingsAccount', 'Account', 'object']
```

The precise algorithm for finding method resolution orderings is not a topic for this course, but is described by Python's primary author with a reference to the original paper.

2.5.8 The Role of Objects

The Python object system is designed to make data abstraction and message passing both convenient and flexible. The specialized syntax of classes, methods, inheritance, and dot expressions all enable us to formalize the object metaphor in our programs, which improves our ability to organize large programs.

In particular, we would like our object system to promote a *separation of concerns* among the different aspects of the program. Each object in a program encapsulates and manages some part of the program's state, and each class statement defines the functions that implement some part of the program's overall logic. Abstraction barriers enforce the boundaries between different aspects of a large program.

Object-oriented programming is particularly well-suited to programs that model systems that have separate but interacting parts. For instance, different users interact in a social network, different characters interact in a game, and different shapes interact in a physical simulation. When representing such systems, the objects in a program often map naturally onto objects in the system being modeled, and classes represent their types and relationships.

On the other hand, classes may not provide the best mechanism for implementing certain abstractions. Functional abstractions provide a more natural metaphor for representing relationships between inputs and outputs. One should not feel compelled to fit every bit of logic in a program within a class, especially when defining independent functions for manipulating data is more natural. Functions can also enforce a separation of concerns.

Multi-paradigm languages like Python allow programmers to match organizational paradigms to appropriate problems. Learning to identify when to introduce a new class, as opposed to a new function, in order to simplify or modularize a program, is an important design skill in software engineering that deserves careful attention.

2.6 Implementing Classes and Objects

When working in the object-oriented programming paradigm, we use the object metaphor to guide the organization of our programs. Most logic about how to represent and manipulate data is expressed within class declarations. In this section, we see that classes and objects can themselves be represented using just functions and dictionaries. The purpose of implementing an object system in this way is to illustrate that using the object metaphor does not require a special programming language. Programs can be object-oriented, even in programming languages that do not have a built-in object system.

In order to implement objects, we will abandon dot notation (which does require built-in language support), but create dispatch dictionaries that behave in much the same way as the elements of the built-in object system. We have already seen how to implement message-passing behavior through dispatch dictionaries. To implement an object system in full, we send messages between instances, classes, and base classes, all of which are dictionaries that contain attributes.

We will not implement the entire Python object system, which includes features that we have not covered in this text (e.g., meta-classes and static methods). We will focus instead on user-defined classes without multiple inheritance and without introspective behavior (such as returning the class of an instance). Our implementation is not meant to follow the precise specification of the Python type system. Instead, it is designed to implement the core functionality that enables the object metaphor.

2.6.1 Instances

We begin with instances. An instance has named attributes, such as the balance of an account, which can be set and retrieved. We implement an instance using a dispatch dictionary that responds to messages that "get" and "set" attribute values. Attributes themselves are stored in a local dictionary called `attributes`.

As we have seen previously in this chapter, dictionaries themselves are abstract data types. We implemented dictionaries with lists, we implemented lists with pairs, and we implemented pairs with functions. As we implement an object system in terms of dictionaries, keep in mind that we could just as well be implementing objects using functions alone.

To begin our implementation, we assume that we have a class implementation that can look up any names that are not part of the instance. We pass in a class to `make_instance` as the parameter `cls`.

```
>>> def make_instance(cls):
        """Return a new object instance, which is a dispatch dictionary."""
        def get_value(name):
            if name in attributes:
                return attributes[name]
            else:
                value = cls['get'](name)
                return bind_method(value, instance)
        def set_value(name, value):
            attributes[name] = value
        attributes = {}
        instance = {'get': get_value, 'set': set_value}
        return instance
```

The `instance` is a dispatch dictionary that responds to the messages `get` and `set`. The `set` message corresponds to attribute assignment in Python's object system: all assigned attributes are stored directly within the object's local attribute dictionary. In `get`, if `name` does not appear in the local `attributes` dictionary, then it is looked up in the class. If the `value` returned by `cls` is a function, it must be bound to the instance.

Bound method values. The `get_value` function in `make_instance` finds a named attribute in its class with `get`, then calls `bind_method`. Binding a method only applies to function values, and it creates a bound method value from a function value by inserting the instance as the first argument:

```
>>> def bind_method(value, instance):
        """Return a bound method if value is callable, or value otherwise."""
        if callable(value):
            def method(*args):
                return value(instance, *args)
            return method
        else:
            return value
```

When a method is called, the first parameter `self` will be bound to the value of `instance` by this definition.

2.6.2 Classes

A class is also an object, both in Python's object system and the system we are implementing here. For simplicity, we say that classes do not themselves have a class. (In Python, classes do have classes; almost all classes share the same class, called `type`.) A class can respond to `get` and `set` messages, as well as the `new` message:

```
>>> def make_class(attributes, base_class=None):
        """Return a new class, which is a dispatch dictionary."""
        def get_value(name):
            if name in attributes:
                return attributes[name]
            elif base_class is not None:
                return base_class['get'](name)
        def set_value(name, value):
            attributes[name] = value
        def new(*args):
            return init_instance(cls, *args)
        cls = {'get': get_value, 'set': set_value, 'new': new}
        return cls
```

Unlike an instance, the `get` function for classes does not query its class when an attribute is not found, but instead queries its `base_class`. No method binding is required for classes.

Initialization. The `new` function in `make_class` calls `init_instance`, which first makes a new instance, then invokes a method called `__init__`.

```
>>> def init_instance(cls, *args):
        """Return a new object with type cls, initialized with args."""
        instance = make_instance(cls)
        init = cls['get']('__init__')
        if init:
            init(instance, *args)
        return instance
```

This final function completes our object system. We now have instances, which `set` locally but fall back to their classes on `get`. After an instance looks up a name in its class, it binds itself to function values to create methods. Finally, classes can create `new` instances, and they apply their `__init__` constructor function immediately after instance creation.

In this object system, the only function that should be called by the user is `create_class`. All other functionality is enabled through message passing. Similarly, Python's object system is invoked via the `class` statement, and all of its other functionality is enabled through dot expressions and calls to classes.

2.6.3 Using Implemented Objects

We now return to use the bank account example from the previous section. Using our implemented object system, we will create an `Account` class, a `CheckingAccount` subclass, and an instance of each.

The `Account` class is created through a `create_account_class` function, which has structure similar to a `class` statement in Python, but concludes with a call to `make_class`.

```
>>> def make_account_class():
        """Return the Account class, which has deposit and withdraw methods."""
        def __init__(self, account_holder):
            self['set']('holder', account_holder)
            self['set']('balance', 0)
        def deposit(self, amount):
            """Increase the account balance by amount and return the new balance."""
            new_balance = self['get']('balance') + amount
            self['set']('balance', new_balance)
            return self['get']('balance')
        def withdraw(self, amount):
            """Decrease the account balance by amount and return the new balance."""
            balance = self['get']('balance')
            if amount > balance:
                return 'Insufficient funds'
            self['set']('balance', balance - amount)
            return self['get']('balance')
        return make_class({'__init__': __init__,
                           'deposit':  deposit,
                           'withdraw': withdraw,
                           'interest': 0.02})
```

In this function, the names of attributes are set at the end. Unlike Python `class` statements, which enforce consistency between intrinsic function names and attribute names, here we must specify the correspondence between attribute names and values manually.

The `Account` class is finally instantiated via assignment.

```
>>> Account = make_account_class()
```

Then, an account instance is created via the `new` message, which requires a name to go with the newly created account.

```
>>> jim_acct = Account['new']('Jim')
```

Then, `get` messages passed to `jim_acct` retrieve properties and methods. Methods can be called to update the balance of the account.

```
>>> jim_acct['get']('holder')
'Jim'
>>> jim_acct['get']('interest')
0.02
>>> jim_acct['get']('deposit')(20)
20
>>> jim_acct['get']('withdraw')(5)
15
```

As with the Python object system, setting an attribute of an instance does not change the corresponding attribute of its class.

```
>>> jim_acct['set']('interest', 0.04)
>>> Account['get']('interest')
0.02
```

Inheritance. We can create a subclass `CheckingAccount` by overloading a subset of the class attributes. In this case, we change the `withdraw` method to impose a fee, and we reduce the interest rate.

```
>>> def make_checking_account_class():
        """Return the CheckingAccount class, which imposes a $1 withdrawal fee."""
        def withdraw(self, amount):
            return Account['get']('withdraw')(self, amount + 1)
        return make_class({'withdraw': withdraw, 'interest': 0.01}, Account)
```

In this implementation, we call the `withdraw` function of the base class `Account` from the `withdraw` function of the subclass, as we would in Python's built-in object system. We can create the subclass itself and an instance, as before.

```
>>> CheckingAccount = make_checking_account_class()
>>> jack_acct = CheckingAccount['new']('Jack')
```

Deposits behave identically, as does the constructor function. withdrawals impose the $1 fee from the specialized `withdraw` method, and `interest` has the new lower value from `CheckingAccount`.

```
>>> jack_acct['get']('interest')
0.01
>>> jack_acct['get']('deposit')(20)
20
>>> jack_acct['get']('withdraw')(5)
14
```

Our object system built upon dictionaries is quite similar in implementation to the built-in object system in Python. In Python, an instance of any user-defined class has a special attribute `__dict__` that stores the local instance attributes for that object in a dictionary, much like our `attributes` dictionary. Python differs because it distinguishes certain special methods that interact with built-in functions to ensure that those functions behave correctly for arguments of many different types. Functions that operate on different types are the subject of the next section.

2.7 Generic Operations

In this chapter, we introduced compound data values, along with the technique of data abstraction using constructors and selectors. Using message passing, we endowed our abstract data types with behavior directly. Using the object metaphor, we bundled together the representation of data and the methods used to manipulate that data to modularize data-driven programs with local state.

However, we have yet to show that our object system allows us to combine together different types of objects flexibly in a large program. Message passing via dot expressions is only one way of building combined expressions with multiple objects. In this section, we explore alternate methods for combining and manipulating objects of different types.

2.7.1 String Conversion

We stated in the beginning of this chapter that an object value should behave like the kind of data it is meant to represent, including producing a string representation of itself. String representations of data values are especially important in an interactive language like Python, where the `read-eval-print` loop requires every value to have some sort of string representation.

String values provide a fundamental medium for communicating information among humans. Sequences of characters can be rendered on a screen, printed to paper, read aloud, converted to braille, or broadcast as Morse code. Strings are also fundamental to programming because they can represent Python expressions. For an object, we may want to generate a string that, when interpreted as a Python expression, evaluates to an equivalent object.

Python stipulates that all objects should produce two different string representations: one that is human-interpretable text and one that is a Python-interpretable expression. The constructor function for strings, `str`, returns a human-readable string. Where possible, the `repr` function returns a Python expression that evaluates to an equal object. The docstring for *repr* explains this property:

```
repr(object) -> string

Return the canonical string representation of the object.
For most object types, eval(repr(object)) == object.
```

The result of calling `repr` on the value of an expression is what Python prints in an interactive session.

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

In cases where no representation exists that evaluates to the original value, Python produces a proxy.

```
>>> repr(min)
'<built-in function min>'
```

The `str` constructor often coincides with `repr`, but provides a more interpretable text representation in some cases. For instance, we see a difference between `str` and `repr` with dates.

```
>>> from datetime import date
>>> today = date(2011, 9, 12)
>>> repr(today)
'datetime.date(2011, 9, 12)'
>>> str(today)
'2011-09-12'
```

Defining the `repr` function presents a new challenge: we would like it to apply correctly to all data types, even those that did not exist when `repr` was implemented. We would like it to be a *polymorphic function*, one that can be applied to many (*poly*) different forms (*morph*) of data.

Message passing provides an elegant solution in this case: the `repr` function invokes a method called `__repr__` on its argument.

```
>>> today.__repr__()
'datetime.date(2011, 9, 12)'
```

By implementing this same method in user-defined classes, we can extend the applicability of `repr` to any class we create in the future. This example highlights another benefit of message passing in general, that it provides a mechanism for extending the domain of existing functions to new object types.

The `str` constructor is implemented in a similar manner: it invokes a method called `__str__` on its argument.

```
>>> today.__str__()
'2011-09-12'
```

These polymorphic functions are examples of a more general principle: certain functions should apply to multiple data types. The message passing approach exemplified here is only one of a family of techniques for implementing polymorphic functions. The remainder of this section explores some alternatives.

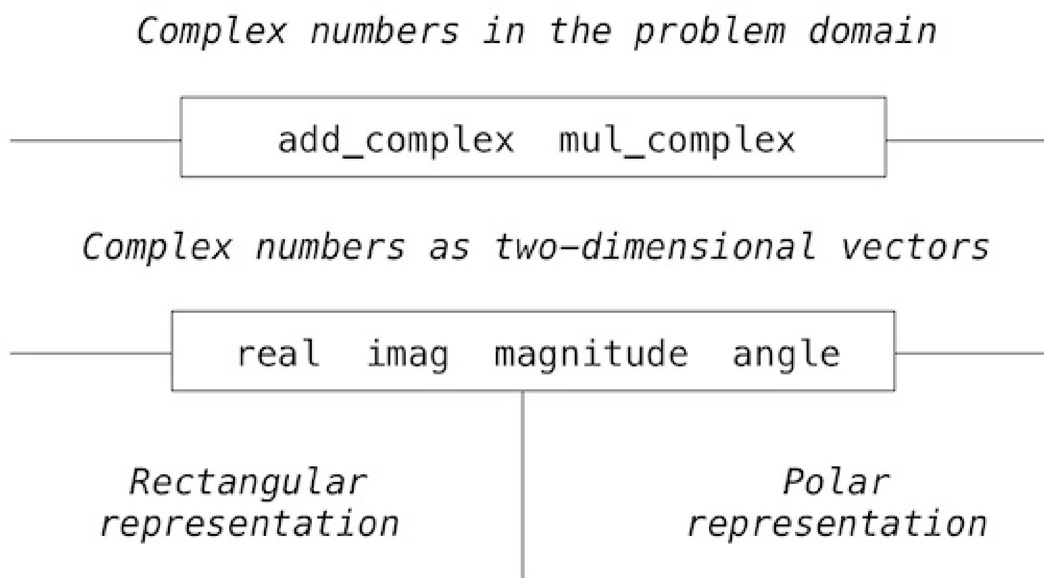
2.7.2 Multiple Representations

Data abstraction, using objects or functions, is a powerful tool for managing complexity. Abstract data types allow us to construct an abstraction barrier between the underlying representation of data and the functions or messages used to manipulate it. However, in large programs, it may not always make sense to speak of "the underlying representation" for a data type in a program. For one thing, there might be more than one useful representation for a data object, and we might like to design systems that can deal with multiple representations.

To take a simple example, complex numbers may be represented in two almost equivalent ways: in rectangular form (real and imaginary parts) and in polar form (magnitude and angle). Sometimes the rectangular form is more appropriate and sometimes the polar form is more appropriate. Indeed, it is perfectly plausible to imagine a system in which complex numbers are represented in both ways, and in which the functions for manipulating complex numbers work with either representation.

More importantly, large software systems are often designed by many people working over extended periods of time, subject to requirements that change over time. In such an environment, it is simply not possible for everyone to agree in advance on choices of data representation. In addition to the data-abstraction barriers that isolate representation from use, we need abstraction barriers that isolate different design choices from each other and permit different choices to coexist in a single program. Furthermore, since large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems additively, that is, without having to redesign or re-implement these modules.

We begin with the simple complex-number example. We will see how message passing enables us to design separate rectangular and polar representations for complex numbers while maintaining the notion of an abstract "complex-number" object. We will accomplish this by defining arithmetic functions for complex numbers (`add_complex` , `mul_complex`) in terms of generic selectors that access parts of a complex number independent of how the number is represented. The resulting complex-number system contains two different kinds of abstraction barriers. They isolate higher-level operations from lower-level representations. In addition, there is a vertical barrier that gives us the ability to separately design alternative representations.



As a side note, we are developing a system that performs arithmetic operations on complex numbers as a simple but unrealistic example of a program that uses generic operations. A complex number type is actually built into Python, but for this example we will implement our own.

Like rational numbers, complex numbers are naturally represented as pairs. The set of complex numbers can be thought of as a two-dimensional space with two orthogonal axes, the real axis and the imaginary axis. From this point of view, the complex number `z = x + y * i` (where `i*i = -1`) can be thought of as the point in the plane whose real coordinate is `x` and whose imaginary coordinate is `y`. Adding complex numbers involves adding their respective `x` and `y` coordinates.

When multiplying complex numbers, it is more natural to think in terms of representing a complex number in polar form, as a magnitude and an angle. The product of two complex numbers is the vector obtained by stretching one complex number by a factor of the length of the other, and then rotating it through the angle of the other.

Thus, there are two different representations for complex numbers, which are appropriate for different operations. Yet, from the viewpoint of someone writing a program that uses complex numbers, the principle of data abstraction suggests that all the operations for manipulating complex numbers should be available regardless of which representation is used by the computer.

Interfaces. Message passing not only provides a method for coupling behavior and data, it allows different data types to respond to the same message in different ways. A shared message that elicits similar behavior from different object classes is a powerful method of abstraction.

As we have seen, an abstract data type is defined by constructors, selectors, and additional behavior conditions. A closely related concept is an *interface*, which is a set of shared messages, along with a specification of what they mean. Objects that respond to the special `__repr__` and `__str__` methods all implement a common interface of types that can be represented as strings.

In the case of complex numbers, the interface needed to implement arithmetic consists of four messages: `real` , `imag` , `magnitude` , and `angle` . We can implement addition and multiplication in terms of these messages.

We can have two different abstract data types for complex numbers that differ in their constructors.

- `ComplexRI` constructs a complex number from real and imaginary parts.
- `ComplexMA` constructs a complex number from a magnitude and angle.

With these messages and constructors, we can implement complex arithmetic.

```
>>> def add_complex(z1, z2):
        return ComplexRI(z1.real + z2.real, z1.imag + z2.imag)
```

```
>>> def mul_complex(z1, z2):
        return ComplexMA(z1.magnitude * z2.magnitude, z1.angle + z2.angle)
```

The relationship between the terms "abstract data type" (ADT) and "interface" is subtle. An ADT includes ways of building complex data types, manipulating them as units, and selecting for their components. In an object-oriented system, an ADT corresponds to a class, although we have seen that an object system is not needed to implement an ADT. An interface is a set of messages that have associated meanings, and which may or may not include selectors. Conceptually, an ADT describes a full representational abstraction of some kind of thing, whereas an interface specifies a set of behaviors that may be shared across many things.

Properties. We would like to use both types of complex numbers interchangeably, but it would be wasteful to store redundant information about each number. We would like to store either the real-imaginary representation or the magnitude-angle representation.

Python has a simple feature for computing attributes on the fly from zero-argument functions. The `@property` decorator allows functions to be called without the standard call expression syntax. An implementation of complex numbers in terms of real and imaginary parts illustrates this point.

```
>>> from math import atan2
>>> class ComplexRI(object):
        def __init__(self, real, imag):
            self.real = real
            self.imag = imag
        @property
        def magnitude(self):
            return (self.real ** 2 + self.imag ** 2) ** 0.5
        @property
        def angle(self):
            return atan2(self.imag, self.real)
        def __repr__(self):
            return 'ComplexRI({0}, {1})'.format(self.real, self.imag)
```

A second implementation using magnitude and angle provides the same interface because it responds to the same set of messages.

```
>>> from math import sin, cos
>>> class ComplexMA(object):
        def __init__(self, magnitude, angle):
            self.magnitude = magnitude
            self.angle = angle
        @property
        def real(self):
            return self.magnitude * cos(self.angle)
        @property
        def imag(self):
            return self.magnitude * sin(self.angle)
        def __repr__(self):
            return 'ComplexMA({0}, {1})'.format(self.magnitude, self.angle)
```

In fact, our implementations of `add_complex` and `mul_complex` are now complete; either class of complex number can be used for either argument in either complex arithmetic function. It is worth noting that the object system does not explicitly connect the two complex types in any way (e.g., through inheritance). We have implemented the complex number abstraction by sharing a common set of messages, an interface, across the two classes.

```
>>> from math import pi
>>> add_complex(ComplexRI(1, 2), ComplexMA(2, pi/2))
ComplexRI(1.0000000000000002, 4.0)
>>> mul_complex(ComplexRI(0, 1), ComplexRI(0, 1))
ComplexMA(1.0, 3.141592653589793)
```

The interface approach to encoding multiple representations has appealing properties. The class for each representation can be developed separately; they must only agree on the names of the attributes they share. The interface is also *additive*. If another programmer wanted to add a third representation of complex numbers to the same program, they would only have to create another class with the same attributes.

Special methods. The built-in mathematical operators can be extended in much the same way as `repr`; there are special method names corresponding to Python operators for arithmetic, logical, and sequence operations.

To make our code more legible, we would perhaps like to use the `+` and `*` operators directly when adding and multiplying complex numbers. Adding the following methods to both of our complex number classes will enable these operators to be used, as well as the `add` and `mul` functions in the `operator` module:

```
>>> ComplexRI.__add__ = lambda self, other: add_complex(self, other)
>>> ComplexMA.__add__ = lambda self, other: add_complex(self, other)
>>> ComplexRI.__mul__ = lambda self, other: mul_complex(self, other)
>>> ComplexMA.__mul__ = lambda self, other: mul_complex(self, other)
```

Now, we can use infix notation with our user-defined classes.

```
>>> ComplexRI(1, 2) + ComplexMA(2, 0)
ComplexRI(3.0, 2.0)
>>> ComplexRI(0, 1) * ComplexRI(0, 1)
ComplexMA(1.0, 3.141592653589793)
```

Further reading. To evaluate expressions that contain the `+` operator, Python checks for special methods on both the left and right operands of the expression. First, Python checks for an `__add__` method on the value of the left operand, then checks for an `__radd__` method on the value of the right operand. If either is found, that method is invoked with the value of the other operand as its argument.

Similar protocols exist for evaluating expressions that contain any kind of operator in Python, including slice notation and Boolean operators. The Python docs list the exhaustive set of method names for operators. Dive into Python 3 has a chapter on special method names that describes many details of their use in the Python interpreter.

2.7.3 Generic Functions

Our implementation of complex numbers has made two data types interchangeable as arguments to the `add_complex` and `mul_complex` functions. Now we will see how to use this same idea not only to define operations that are generic over different representations but also to define operations that are generic over different kinds of arguments that do not share a common interface.

The operations we have defined so far treat the different data types as being completely independent. Thus, there are separate packages for adding, say, two rational numbers, or two complex numbers. What we have not yet considered is the fact that it is meaningful to define operations that cross the type boundaries, such as the addition of a complex number to a rational number. We have gone to great pains to introduce barriers between parts of our programs so that they can be developed and understood separately.

We would like to introduce the cross-type operations in some carefully controlled way, so that we can support them without seriously violating our abstraction boundaries. There is a tension between the outcomes we desire: we would like to be able to add a complex number to a rational number, and we would like to do so using a generic `add` function that does the right thing with all numeric types. At the same time, we would like to separate the concerns of complex numbers and rational numbers whenever possible, in order to maintain a modular program.

Let us revise our implementation of rational numbers to use Python's built-in object system. As before, we will store a rational number as a numerator and denominator in lowest terms.

```
>>> from fractions import gcd
>>> class Rational(object):
        def __init__(self, numer, denom):
            g = gcd(numer, denom)
            self.numer = numer // g
            self.denom = denom // g
        def __repr__(self):
            return 'Rational({0}, {1})'.format(self.numer, self.denom)
```

Adding and multiplying rational numbers in this new implementation is similar to before.

```
>>> def add_rational(x, y):
        nx, dx = x.numer, x.denom
        ny, dy = y.numer, y.denom
        return Rational(nx * dy + ny * dx, dx * dy)
```

```
>>> def mul_rational(x, y):
        return Rational(x.numer * y.numer, x.denom * y.denom)
```

Type dispatching. One way to handle cross-type operations is to design a different function for each possible combination of types for which the operation is valid. For example, we could extend our complex number implementation so that it provides a function for adding complex numbers to rational numbers. We can provide this functionality generically using a technique called *dispatching on type*.

The idea of type dispatching is to write functions that first inspect the type of argument they have received, and then execute code that is appropriate for the type. In Python, the type of an object can be inspected with the built-in `type` function.

```
>>> def iscomplex(z):
        return type(z) in (ComplexRI, ComplexMA)
```

```
>>> def isrational(z):
        return type(z) == Rational
```

In this case, we are relying on the fact that each object knows its type, and we can look up that type using the Python `type` function. Even if the `type` function were not available, we could imagine implementing `iscomplex` and `isrational` in terms of a shared class attribute for `Rational` , `ComplexRI` , and `ComplexMA` .

Now consider the following implementation of `add` , which explicitly checks the type of both arguments. We will not use Python's special methods (i.e., `__add__`) in this example.

```
>>> def add_complex_and_rational(z, r):
            return ComplexRI(z.real + r.numer/r.denom, z.imag)
```

```
>>> def add(z1, z2):
        """Add z1 and z2, which may be complex or rational."""
        if iscomplex(z1) and iscomplex(z2):
            return add_complex(z1, z2)
        elif iscomplex(z1) and isrational(z2):
            return add_complex_and_rational(z1, z2)
        elif isrational(z1) and iscomplex(z2):
            return add_complex_and_rational(z2, z1)
        else:
            return add_rational(z1, z2)
```

This simplistic approach to type dispatching, which uses a large conditional statement, is not additive. If another numeric type were included in the program, we would have to re-implement `add` with new clauses.

We can create a more flexible implementation of `add` by implementing type dispatch through a dictionary. The first step in extending the flexibility of `add` will be to create a tag set for our classes that abstracts away from the two implementations of complex numbers.

```
>>> def type_tag(x):
        return type_tag.tags[type(x)]
```

```
>>> type_tag.tags = {ComplexRI: 'com', ComplexMA: 'com', Rational: 'rat'}
```

Next, we use these type tags to index a dictionary that stores the different ways of adding numbers. The keys of the dictionary are tuples of type tags, and the values are type-specific addition functions.

```
>>> def add(z1, z2):
        types = (type_tag(z1), type_tag(z2))
        return add.implementations[types](z1, z2)
```

This definition of `add` does not have any functionality itself; it relies entirely on a dictionary called `add.implementations` to implement addition. We can populate that dictionary as follows.

```
>>> add.implementations = {}
>>> add.implementations[('com', 'com')] = add_complex
>>> add.implementations[('com', 'rat')] = add_complex_and_rational
>>> add.implementations[('rat', 'com')] = lambda x, y: add_complex_and_rational(y, x)
>>> add.implementations[('rat', 'rat')] = add_rational
```

This dictionary-based approach to dispatching is additive, because `add.implementations` and `type_tag.tags` can always be extended. Any new numeric type can "install" itself into the existing system by adding new entries to these dictionaries.

While we have introduced some complexity to the system, we now have a generic, extensible `add` function that handles mixed types.

```
>>> add(ComplexRI(1.5, 0), Rational(3, 2))
ComplexRI(3.0, 0)
>>> add(Rational(5, 3), Rational(1, 2))
Rational(13, 6)
```

Data-directed programming. Our dictionary-based implementation of `add` is not addition-specific at all; it does not contain any direct addition logic. It only implements addition because we happen to have populated its `implementations` dictionary with functions that perform addition.

A more general version of generic arithmetic would apply arbitrary operators to arbitrary types and use a dictionary to store implementations of various combinations. This fully generic approach to implementing methods is called *data-directed programming*. In our case, we can implement both generic addition and multiplication without redundant logic.

```
>>> def apply(operator_name, x, y):
        tags = (type_tag(x), type_tag(y))
        key = (operator_name, tags)
        return apply.implementations[key](x, y)
```

In this generic `apply` function, a key is constructed from the operator name (e.g., `'add'`) and a tuple of type tags for the arguments. Implementations are also populated using these tags. We enable support for multiplication on complex and rational numbers below.

```
>>> def mul_complex_and_rational(z, r):
        return ComplexMA(z.magnitude * r.numer / r.denom, z.angle)
```

```
>>> mul_rational_and_complex = lambda r, z: mul_complex_and_rational(z, r)
>>> apply.implementations = {('mul', ('com', 'com')): mul_complex,
                             ('mul', ('com', 'rat')): mul_complex_and_rational,
                             ('mul', ('rat', 'com')): mul_rational_and_complex,
                             ('mul', ('rat', 'rat')): mul_rational}
```

We can also include the addition implementations from `add` to `apply`, using the dictionary `update` method.

```
>>> adders = add.implementations.items()
>>> apply.implementations.update({('add', tags):fn for (tags, fn) in adders})
```

Now that apply supports 8 different implementations in a single table, we can use it to manipulate rational and complex numbers quite generically.

```
>>> apply('add', ComplexRI(1.5, 0), Rational(3, 2))
ComplexRI(3.0, 0)
>>> apply('mul', Rational(1, 2), ComplexMA(10, 1))
ComplexMA(5.0, 1)
```

This data-directed approach does manage the complexity of cross-type operators, but it is cumbersome. With such a system, the cost of introducing a new type is not just writing methods for that type, but also the construction and installation of the functions that implement the cross-type operations. This burden can easily require much more code than is needed to define the operations on the type itself.

While the techniques of dispatching on type and data-directed programming do create additive implementations of generic functions, they do not effectively separate implementation concerns; implementors of the individual numeric types need to take account of other types when writing cross-type operations. Combining rational numbers and complex numbers isn't strictly the domain of either type. Formulating coherent policies on the division of responsibility among types can be an overwhelming task in designing systems with many types and cross-type operations.

Coercion. In the general situation of completely unrelated operations acting on completely unrelated types, implementing explicit cross-type operations, cumbersome though it may be, is the best that one can hope for. Fortunately, we can sometimes do better by taking advantage of additional structure that may be latent in our type system. Often the different data types are not completely independent, and there may be ways by which objects of one type may be viewed as being of another type. This process is called *coercion*. For example, if we are asked to arithmetically combine a rational number with a complex number, we can view the rational number as a complex number whose imaginary part is zero. By doing so, we transform the problem to that of combining two complex numbers, which can be handled in the ordinary way by `add_complex` and `mul_complex`.

In general, we can implement this idea by designing coercion functions that transform an object of one type into an equivalent object of another type. Here is a typical coercion function, which transforms a rational number to a complex number with zero imaginary part:

```
>>> def rational_to_complex(x):
        return ComplexRI(x.numer/x.denom, 0)
```

Now, we can define a dictionary of coercion functions. This dictionary could be extended as more numeric types are introduced.

```
>>> coercions = {('rat', 'com'): rational_to_complex}
```

It is not generally possible to coerce an arbitrary data object of each type into all other types. For example, there is no way to coerce an arbitrary complex number to a rational number, so there will be no such conversion implementation in the `coercions` dictionary.

Using the `coercions` dictionary, we can write a function called `coerce_apply`, which attempts to coerce arguments into values of the same type, and only then applies an operator. The implementations dictionary of `coerce_apply` does not include any cross-type operator implementations.

```
>>> def coerce_apply(operator_name, x, y):
        tx, ty = type_tag(x), type_tag(y)
        if tx != ty:
            if (tx, ty) in coercions:
                tx, x = ty, coercions[(tx, ty)](x)
            elif (ty, tx) in coercions:
                ty, y = tx, coercions[(ty, tx)](y)
            else:
                return 'No coercion possible.'
        key = (operator_name, tx)
        return coerce_apply.implementations[key](x, y)
```

The `implementations` of `coerce_apply` require only one type tag, because they assume that both values share the same type tag. Hence, we require only four implementations to support generic arithmetic over complex and rational numbers.

```
>>> coerce_apply.implementations = {('mul', 'com'): mul_complex,
                                    ('mul', 'rat'): mul_rational,
                                    ('add', 'com'): add_complex,
                                    ('add', 'rat'): add_rational}
```

With these implementations in place, `coerce_apply` can replace `apply`.

```
>>> coerce_apply('add', ComplexRI(1.5, 0), Rational(3, 2))
ComplexRI(3.0, 0)
>>> coerce_apply('mul', Rational(1, 2), ComplexMA(10, 1))
ComplexMA(5.0, 1.0)
```

This coercion scheme has some advantages over the method of defining explicit cross-type operations. Although we still need to write coercion functions to relate the types, we need to write only one function for each pair of types rather than a different functions for each

collection of types and each generic operation. What we are counting on here is the fact that the appropriate transformation between types depends only on the types themselves, not on the particular operation to be applied.

Further advantages come from extending coercion. Some more sophisticated coercion schemes do not just try to coerce one type into another, but instead may try to coerce two different types each into a third common type. Consider a rhombus and a rectangle: neither is a special case of the other, but both can be viewed as quadrilaterals. Another extension to coercion is iterative coercion, in which one data type is coerced into another via intermediate types. Consider that an integer can be converted into a real number by first converting it into a rational number, then converting that rational number into a real number. Chaining coercion in this way can reduce the total number of coercion functions that are required by a program.

Despite its advantages, coercion does have potential drawbacks. For one, coercion functions can lose information when they are applied. In our example, rational numbers are exact representations, but become approximations when they are converted to complex numbers.

Some programming languages have automatic coercion systems built in. In fact, early versions of Python had a `__coerce__` special method on objects. In the end, the complexity of the built-in coercion system did not justify its use, and so it was removed. Instead, particular operators apply coercion to their arguments as needed. Operators are implemented as method calls on user defined types using special methods like `__add__` and `__mul__`. It is left up to you, the user, to decide whether to employ type dispatching, data-directed programming, message passing, or coercion in order to implement generic functions in your programs.

Chapter 3: The Structure and Interpretation of Computer Programs

Contents

3.1 Introduction

Chapters 1 and 2 describe the close connection between two fundamental elements of programming: functions and data. We saw how functions can be manipulated as data using higher-order functions. We also saw how data can be endowed with behavior using message passing and an object system. We have also studied techniques for organizing large programs, such as functional abstraction, data abstraction, class inheritance, and generic functions. These core concepts constitute a strong foundation upon which to build modular, maintainable, and extensible programs.

This chapter focuses on the third fundamental element of programming: programs themselves. A Python program is just a collection of text. Only through the process of interpretation do we perform any meaningful computation based on that text. A programming language like Python is useful because we can define an *interpreter*, a program that carries out Python's evaluation and execution procedures. It is no exaggeration to regard this as the most fundamental idea in programming, that an interpreter, which determines the meaning of expressions in a programming language, is just another program.

To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others.

3.1.1 Programming Languages

In fact, we can regard many programs as interpreters for some language. For example, the constraint propagator from the previous chapter has its own primitives and means of combination. The constraint language was quite specialized: it provided a declarative method for describing a certain class of mathematical relations, not a fully general language for describing computation. While we have been designing languages of a sort already, the material of this chapter will greatly expand the range of languages we can interpret.

Programming languages vary widely in their syntactic structures, features, and domain of application. Among general purpose programming languages, the constructs of function definition and function application are pervasive. On the other hand, powerful languages exist that do not include an object system, higher-order functions, or even control constructs like `while` and `for` statements. To illustrate just how different languages can be, we will introduce Logo as an example of a powerful and expressive programming language that includes very few advanced features.

In this chapter, we study the design of interpreters and the computational processes that they create when executing programs. The prospect of designing an interpreter for a general programming language may seem daunting. After all, interpreters are programs that can carry out any possible computation, depending on their input. However, typical interpreters have an elegant common structure: two mutually recursive functions. The first evaluates expressions in environments; the second applies functions to arguments.

These functions are *recursive* in that they are defined in terms of each other: applying a function requires evaluating the expressions in its body, while evaluating an expression may involve applying one or more functions. The next two sections of this chapter focus on recursive functions and data structures, which will prove essential to understanding the design of an interpreter. The end of the chapter focuses on two new languages and the task of implementing interpreters for them.

3.2 Functions and the Processes They Generate

A function is a pattern for the *local evolution* of a computational process. It specifies how each stage of the process is built upon the previous stage. We would like to be able to make statements about the overall behavior of a process whose local evolution has been specified by one or more functions. This analysis is very difficult to do in general, but we can at least try to describe some typical patterns of process evolution.

In this section we will examine some common "shapes" for processes generated by simple functions. We will also investigate the rates at which these processes consume the important computational resources of time and space.

3.2.1 Recursive Functions

A function is called *recursive* if the body of that function calls the function itself, either directly or indirectly. That is, the process of executing the body of a recursive function may in turn require applying that function again. Recursive functions do not require any special syntax in Python, but they do require some care to define correctly.

As an introduction to recursive functions, we begin with the task of converting an English word into its Pig Latin equivalent. Pig Latin is a secret language: one that applies a simple, deterministic transformation to each word that veils the meaning of the word. Thomas Jefferson was supposedly an early adopter. The Pig Latin equivalent of an English word moves the initial consonant cluster (which may be empty) from the beginning of the word to the end and follows it by the "-ay" vowel. Hence, the word "pun" becomes "unpay", "stout" becomes "outstay", and "all" becomes "allay".

```
>>> def pig_latin(w):
        """Return the Pig Latin equivalent of English word w."""
        if starts_with_a_vowel(w):
            return w + 'ay'
        return pig_latin(w[1:] + w[0])
```

```
>>> def starts_with_a_vowel(w):
        """Return whether w begins with a vowel."""
        return w[0].lower() in 'aeiou'
```

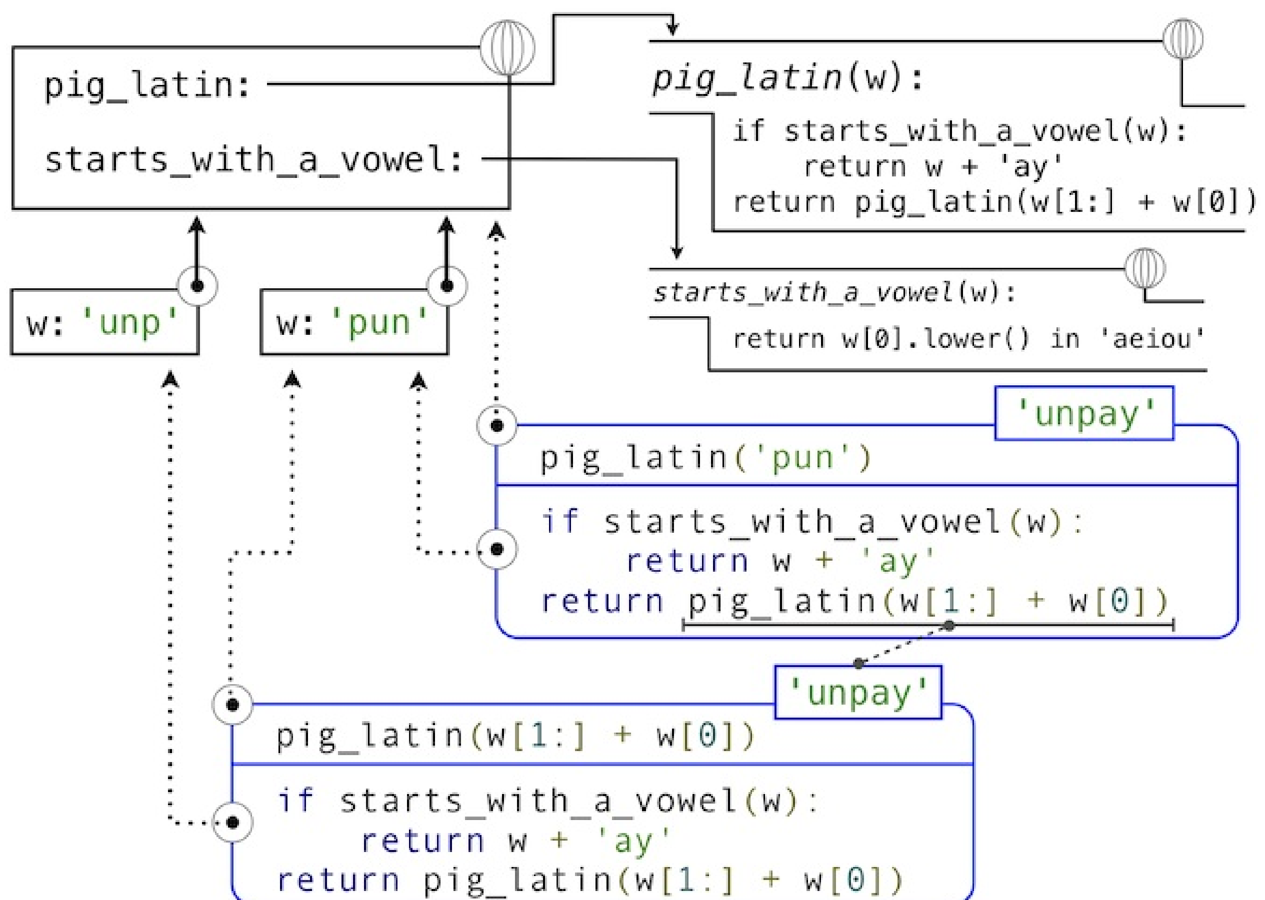
The idea behind this definition is that the Pig Latin variant of a string that starts with a consonant is the same as the Pig Latin variant of another string: that which is created by moving the first letter to the end. Hence, the Pig Latin word for "sending" is the same as for

"endings" (*endingsay*), and the Pig Latin word for "smother" is the same as the Pig Latin word for "mothers" (*othersmay*). Moreover, moving one consonant from the beginning of the word to the end results in a simpler problem with fewer initial consonants. In the case of "sending", moving the "s" to the end gives a word that starts with a vowel, and so our work is done.

This definition of `pig_latin` is both complete and correct, even though the `pig_latin` function is called within its own body.

```
>>> pig_latin('pun')
'unpay'
```

The idea of being able to define a function in terms of itself may be disturbing; it may seem unclear how such a "circular" definition could make sense at all, much less specify a well-defined process to be carried out by a computer. We can, however, understand precisely how this recursive function applies successfully using our environment model of computation. The environment diagram and expression tree that depict the evaluation of `pig_latin('pun')` appear below.



The steps of the Python evaluation procedures that produce this result are:

1. The `def` statement for `pig_latin` is executed, which

> 1. Creates a new *pig_latin* function object with the stated body, and > 2. Binds the name `pig_latin` to that function in the current (global) frame

1. The `def` statement for `starts_with_a_vowel` is executed similarly
2. The call expression `pig_latin('pun')` is evaluated by

> 1. Evaluating the operator and operand sub-expressions by > > > 1. Looking up the name `pig_latin` that is bound to the *pig_latin* function > > 2. Evaluating the operand string literal to the string object `'pun'` > > 1. Applying the function *pig_latin* to the argument `'pun'` by > > > 1. Adding a local frame that extends the global frame > > 2. Binding the formal parameter `w` to the argument `'pun'` in that frame > > 3. Executing the body of *pig_latin* in the environment that starts with that frame: > > > > > 1. The initial conditional statement has no effect, because the header expression evaluates to `False`. > > > 2. The final return expression `pig_latin(w[1:] + w[0])` is evaluated by > > > > > > > 1. Looking up the name `pig_latin` that is bound to the *pig_latin* function > > > > 2. Evaluating the operand expression to the string object `'unp'` > > > > 3. Applying *pig_latin* to the argument `'unp'`, which returns the desired result from the suite of the conditional statement in the body of *pig_latin*.

As this example illustrates, a recursive function applies correctly, despite its circular character. The *pig_latin* function is applied twice, but with a different argument each time. Although the second call comes from the body of *pig_latin* itself, looking up that function by name succeeds because the name `pig_latin` is bound in the environment before its body is executed.

This example also illustrates how Python's recursive evaluation procedure can interact with a recursive function to evolve a complex computational process with many nested steps, even though the function definition may itself contain very few lines of code.

3.2.2 The Anatomy of Recursive Functions

A common pattern can be found in the body of many recursive functions. The body begins with a *base case*, a conditional statement that defines the behavior of the function for the inputs that are simplest to process. In the case of `pig_latin`, the base case occurs for any argument that starts with a vowel. In this case, there is no work left to be done but return the argument with "ay" added to the end. Some recursive functions will have multiple base cases.

The base cases are then followed by one or more *recursive calls*. Recursive calls require a certain character: they must simplify the original problem. In the case of `pig_latin`, the more initial consonants in `w`, the more work there is left to do. In the recursive call,

`pig_latin(w[1:] + w[0])`, we call `pig_latin` on a word that has one fewer initial consonant -- a simpler problem. Each successive call to `pig_latin` will be simpler still until the base case is reached: a word with no initial consonants.

Recursive functions express computation by simplifying problems incrementally. They often operate on problems in a different way than the iterative approaches that we have used in the past. Consider a function `fact` to compute `n` factorial, where for example `fact(4)` computes ($4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$).

A natural implementation using a `while` statement accumulates the total by multiplying together each positive integer up to `n`.

```
>>> def fact_iter(n):
        total, k = 1, 1
        while k <= n:
            total, k = total * k, k + 1
        return total
```

```
>>> fact_iter(4)
24
```

On the other hand, a recursive implementation of factorial can express `fact(n)` in terms of `fact(n-1)`, a simpler problem. The base case of the recursion is the simplest form of the problem: `fact(1)` is `1`.

```
>>> def fact(n):
        if n == 1:
            return 1
        return n * fact(n-1)
```

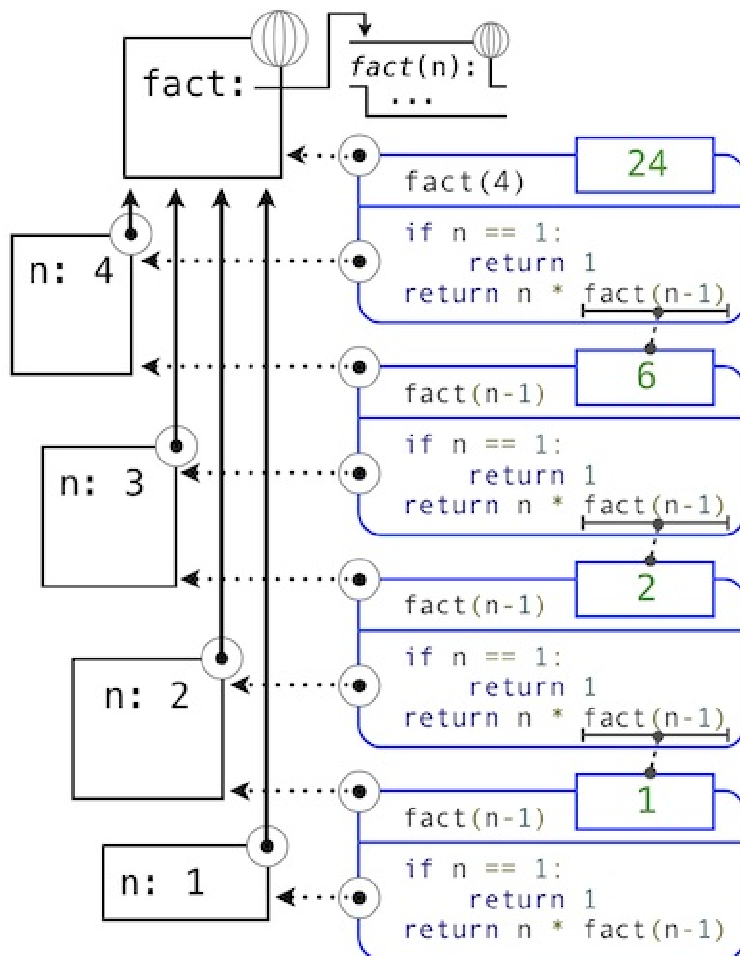
```
>>> fact(4)
24
```

The correctness of this function is easy to verify from the standard definition of the mathematical function for factorial:

$$\begin{array}{l l} (n-1)! &= (n-1) \cdot (n-2) \cdot \dots \cdot 1 \\ n! &= n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1 \\ n! &= n \cdot (n-1)! \end{array}$$

These two factorial functions differ conceptually. The iterative function constructs the result from the base case of 1 to the final total by successively multiplying in each term. The recursive function, on the other hand, constructs the result directly from the final term, `n`, and the result of the simpler problem, `fact(n-1)`.

As the recursion "unwinds" through successive applications of the *fact* function to simpler and simpler problem instances, the result is eventually built starting from the base case. The diagram below shows how the recursion ends by passing the argument `1` to `fact`, and how the result of each call depends on the next until the base case is reached.



While we can unwind the recursion using our model of computation, it is often clearer to think about recursive calls as functional abstractions. That is, we should not care about how `fact(n-1)` is implemented in the body of `fact`; we should simply trust that it computes the factorial of `n-1`. Treating a recursive call as a functional abstraction has been called a *recursive leap of faith*. We define a function in terms of itself, but simply trust that the simpler cases will work correctly when verifying the correctness of the function. In this example, we trust that `fact(n-1)` will correctly compute `(n-1)!`; we must only check that `n!` is computed correctly if this assumption holds. In this way, verifying the correctness of a recursive function is a form of proof by induction.

The functions *fact_iter* and *fact* also differ because the former must introduce two additional names, `total` and `k`, that are not required in the recursive implementation. In general, iterative functions must maintain some local state that changes throughout the course of computation. At any point in the iteration, that state characterizes the result of completed work and the amount of work remaining. For example, when `k` is `3` and total is `2`, there are still two terms remaining to be processed, `3` and `4`. On the other hand, *fact* is

characterized by its single argument `n`. The state of the computation is entirely contained within the structure of the expression tree, which has return values that take the role of `total`, and binds `n` to different values in different frames rather than explicitly tracking `k`.

Recursive functions can rely more heavily on the interpreter itself, by storing the state of the computation as part of the expression tree and environment, rather than explicitly using names in the local frame. For this reason, recursive functions are often easier to define, because we do not need to try to determine the local state that must be maintained across iterations. On the other hand, learning to recognize the computational processes evolved by recursive functions can require some practice.

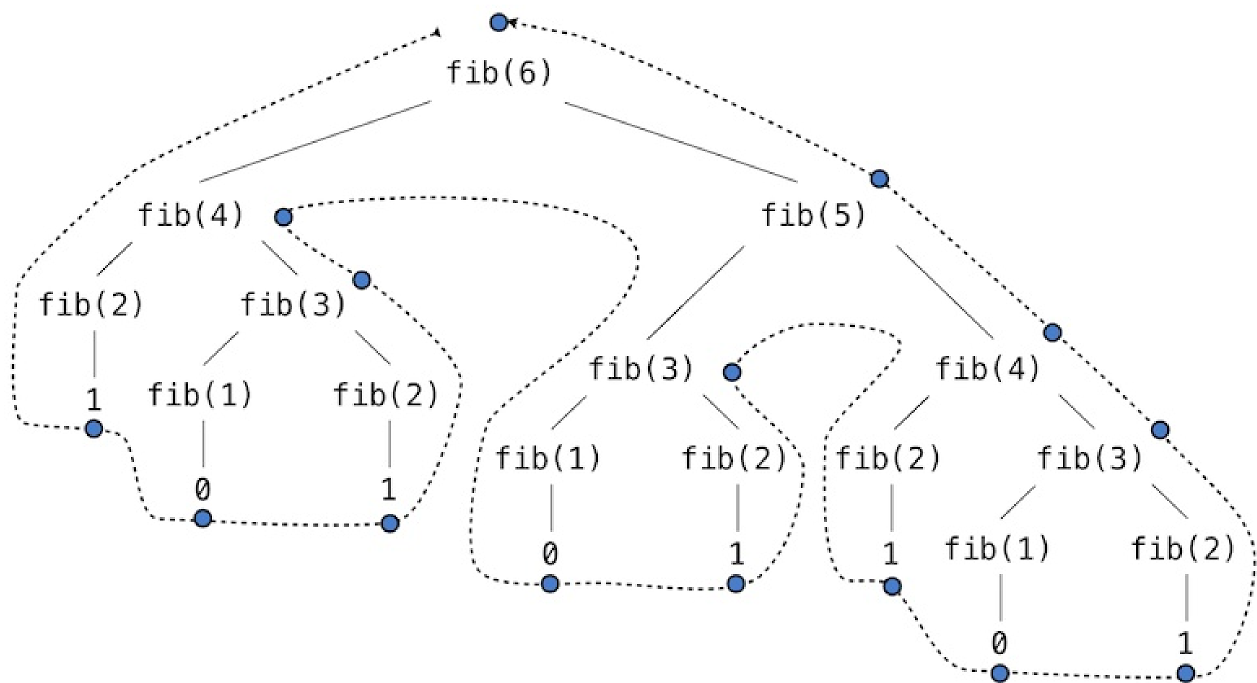
3.2.3 Tree Recursion

Another common pattern of computation is called tree recursion. As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two.

```
>>> def fib(n):
        if n == 1:
            return 0
        if n == 2:
            return 1
        return fib(n-2) + fib(n-1)
```

```
>>> fib(6)
5
```

This recursive definition is tremendously appealing relative to our previous attempts: it exactly mirrors the familiar definition of Fibonacci numbers. Consider the pattern of computation that results from evaluating `fib(6)`, shown below. To compute `fib(6)`, we compute `fib(5)` and `fib(4)`. To compute `fib(5)`, we compute `fib(4)` and `fib(3)`. In general, the evolved process looks like a tree (the diagram below is not a full expression tree, but instead a simplified depiction of the process; a full expression tree would have the same general structure). Each blue dot indicates a completed computation of a Fibonacci number in the traversal of this tree.



Functions that call themselves multiple times in this way are said to be *tree recursive*. This function is instructive as a prototypical tree recursion, but it is a terrible way to compute Fibonacci numbers because it does so much redundant computation. Notice that the entire computation of `fib(4)` -- almost half the work -- is duplicated. In fact, it is not hard to show that the number of times the function will compute `fib(1)` or `fib(2)` (the number of leaves in the tree, in general) is precisely `fib(n+1)`. To get an idea of how bad this is, one can show that the value of `fib(n)` grows exponentially with `n`. Thus, the process uses a number of steps that grows exponentially with the input.

We have already seen an iterative implementation of Fibonacci numbers, repeated here for convenience.

```
>>> def fib_iter(n):
        prev, curr = 1, 0  # curr is the first Fibonacci number.
        for _ in range(n-1):
             prev, curr = curr, prev + curr
        return curr
```

The state that we must maintain in this case consists of the current and previous Fibonacci numbers. Implicitly the `for` statement also keeps track of the iteration count. This definition does not reflect the standard mathematical definition of Fibonacci numbers as clearly as the recursive approach. However, the amount of computation required in the iterative implementation is only linear in `n`, rather than exponential. Even for small values of `n`, this difference can be enormous.

One should not conclude from this difference that tree-recursive processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerful tool. Furthermore, tree-

recursive processes can often be made more efficient.

Memoization. A powerful technique for increasing the efficiency of recursive functions that repeat computation is called *memoization*. A memoized function will store the return value for any arguments it has previously received. A second call to `fib(4)` would not evolve the same complex process as the first, but instead would immediately return the stored result computed by the first call.

Memoization can be expressed naturally as a higher-order function, which can also be used as a decorator. The definition below creates a *cache* of previously computed results, indexed by the arguments from which they were computed. The use of a dictionary will require that the argument to the memoized function be immutable in this implementation.

```
>>> def memo(f):
        """Return a memoized version of single-argument function f."""
        cache = {}
        def memoized(n):
            if n not in cache:
                cache[n] = f(n)
            return cache[n]
        return memoized
```

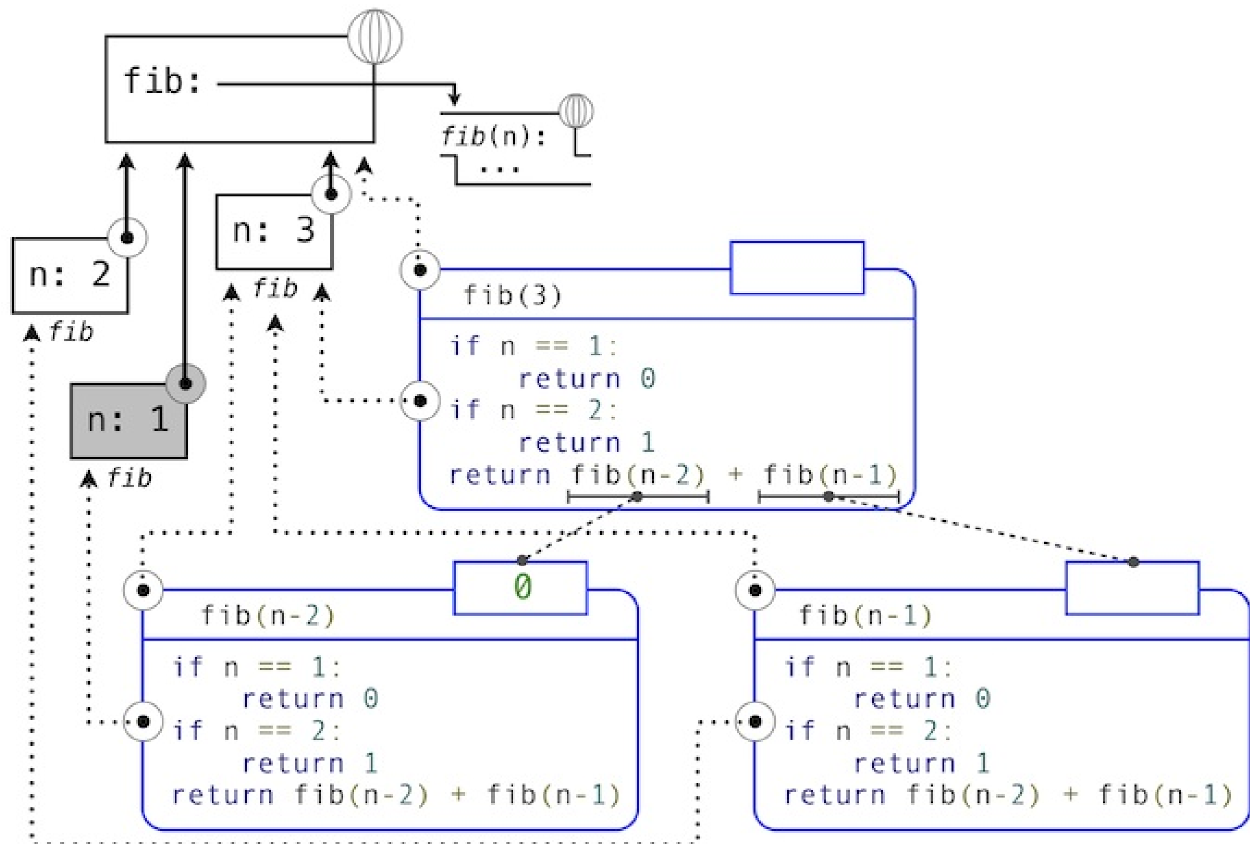
```
>>> fib = memo(fib)
>>> fib(40)
63245986
```

The amount of computation time saved by memoization in this case is substantial. The memoized, recursive *fib* function and the iterative *fib_iter* function both require an amount of time to compute that is only a linear function of their input `n`. To compute `fib(40)`, the body of `fib` is executed 40 times, rather than 102,334,155 times in the unmemoized recursive case.

Space. To understand the space requirements of a function, we must specify generally how memory is used, preserved, and reclaimed in our environment model of computation. In evaluating an expression, we must preserve all *active* environments and all values and frames referenced by those environments. An environment is active if it provides the evaluation context for some expression in the current branch of the expression tree.

For example, when evaluating `fib`, the interpreter proceeds to compute each value in the order shown previously, traversing the structure of the tree. To do so, it only needs to keep track of those nodes that are above the current node in the tree at any point in the computation. The memory used to evaluate the rest of the branches can be reclaimed because it cannot affect future computation. In general, the space required for tree-recursive functions will be proportional to the maximum depth of the tree.

The diagram below depicts the environment and expression tree generated by evaluating `fib(3)`. In the process of evaluating the return expression for the initial application of `fib`, the expression `fib(n-2)` is evaluated, yielding a value of `0`. Once this value is computed, the corresponding environment frame (grayed out) is no longer needed: it is not part of an active environment. Thus, a well-designed interpreter can reclaim the memory that was used to store this frame. On the other hand, if the interpreter is currently evaluating `fib(n-1)`, then the environment created by this application of `fib` (in which `n` is `2`) is active. In turn, the environment originally created to apply `fib` to `3` is active because its value has not yet been successfully computed.



In the case of `memo`, the environment associated with the function it returns (which contains `cache`) must be preserved as long as some name is bound to that function in an active environment. The number of entries in the `cache` dictionary grows linearly with the number of unique arguments passed to `fib`, which scales linearly with the input. On the other hand, the iterative implementation requires only two numbers to be tracked during computation: `prev` and `curr`, giving it a constant size.

Memoization exemplifies a common pattern in programming that computation time can often be decreased at the expense of increased use of space, or vis versa.

3.2.4 Example: Counting Change

Consider the following problem: How many different ways can we make change of $1.00, given half-dollars, quarters, dimes, nickels, and pennies? More generally, can we write a function to compute the number of ways to change any given amount of money using any set of currency denominations?

This problem has a simple solution as a recursive function. Suppose we think of the types of coins available as arranged in some order, say from most to least valuable.

The number of ways to change an amount `a` using `n` kinds of coins equals

1. the number of ways to change `a` using all but the first kind of coin, plus
2. the number of ways to change the smaller amount `a - d` using all `n` kinds of coins, where `d` is the denomination of the first kind of coin.

To see why this is true, observe that the ways to make change can be divided into two groups: those that do not use any of the first kind of coin, and those that do. Therefore, the total number of ways to make change for some amount is equal to the number of ways to make change for the amount without using any of the first kind of coin, plus the number of ways to make change assuming that we do use the first kind of coin at least once. But the latter number is equal to the number of ways to make change for the amount that remains after using a coin of the first kind.

Thus, we can recursively reduce the problem of changing a given amount to the problem of changing smaller amounts using fewer kinds of coins. Consider this reduction rule carefully and convince yourself that we can use it to describe an algorithm if we specify the following base cases:

1. If `a` is exactly `0`, we should count that as `1` way to make change.
2. If `a` is less than `0`, we should count that as `0` ways to make change.
3. If `n` is `0`, we should count that as `0` ways to make change.

We can easily translate this description into a recursive function:

```
>>> def count_change(a, kinds=(50, 25, 10, 5, 1)):
        """Return the number of ways to change amount a using coin kinds."""
        if a == 0:
            return 1
        if a < 0 or len(kinds) == 0:
            return 0
        d = kinds[0]
        return count_change(a, kinds[1:]) + count_change(a - d, kinds)
```

```
>>> count_change(100)
292
```

The `count_change` function generates a tree-recursive process with redundancies similar to those in our first implementation of `fib`. It will take quite a while for that `292` to be computed, unless we memoize the function. On the other hand, it is not obvious how to design an iterative algorithm for computing the result, and we leave this problem as a challenge.

3.2.5 Orders of Growth

The previous examples illustrate that processes can differ considerably in the rates at which they consume the computational resources of space and time. One convenient way to describe this difference is to use the notion of *order of growth* to obtain a coarse measure of the resources required by a process as the inputs become larger.

Let (n) be a parameter that measures the size of the problem, and let (R(n)) be the amount of resources the process requires for a problem of size (n). In our previous examples we took (n) to be the number for which a given function is to be computed, but there are other possibilities. For instance, if our goal is to compute an approximation to the square root of a number, we might take (n) to be the number of digits of accuracy required. In general there are a number of properties of the problem with respect to which it will be desirable to analyze a given process. Similarly, (R(n)) might measure the amount of memory used, the number of elementary machine operations performed, and so on. In computers that do only a fixed number of operations at a time, the time required to evaluate an expression will be proportional to the number of elementary machine operations performed in the process of evaluation.

We say that (R(n)) has order of growth (\Theta(f(n))), written (R(n) = \Theta(f(n))) (pronounced "theta of (f(n))"), if there are positive constants (k_1) and (k_2) independent of (n) such that

[k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)]

for any sufficiently large value of (n). In other words, for large (n), the value (R(n)) is sandwiched between two values that both scale with (f(n)):

- A lower bound (k_1 \cdot f(n)) and
- An upper bound (k_2 \cdot f(n))

For instance, the number of steps to compute (n!) grows proportionally to the input (n). Thus, the steps required for this process grows as (\Theta(n)). We also saw that the space required for the recursive implementation `fact` grows as (\Theta(n)). By contrast, the iterative implementation `fact_iter` takes a similar number of steps, but the space it requires stays constant. In this case, we say that the space grows as (\Theta(1)).

The number of steps in our tree-recursive Fibonacci computation `fib` grows exponentially in its input n. In particular, one can show that the nth Fibonacci number is the closest integer to

$$\frac{\phi^{n-2}}{\sqrt{5}}$$

where ϕ is the golden ratio:

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6180$$

We also stated that the number of steps scales with the resulting value, and so the tree-recursive process requires $\Theta(\phi^n)$ steps, a function that grows exponentially with n.

Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring n^2 steps and a process requiring $1000 \cdot n^2$ steps and a process requiring $3 \cdot n^2 + 10 \cdot n + 17$ steps all have $\Theta(n^2)$ order of growth. There are certainly cases in which an order of growth analysis is too coarse a method for deciding between two possible implementations of a function.

However, order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem. For a $\Theta(n)$ (linear) process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in problem size will multiply the resource utilization by a constant factor. The next example examines an algorithm whose order of growth is logarithmic, so that doubling the problem size increases the resource requirement by only a constant amount.

3.2.6 Example: Exponentiation

Consider the problem of computing the exponential of a given number. We would like a function that takes as arguments a base `b` and a positive integer exponent `n` and computes b^n. One way to do this is via the recursive definition

$$\begin{array}{l l} b^n &= b \cdot b^{n-1} \\ b^0 &= 1 \end{array}$$

which translates readily into the recursive function

```
>>> def exp(b, n):
        if n == 0:
            return 1
        return b * exp(b, n-1)
```

This is a linear recursive process that requires $\Theta(n)$ steps and $\Theta(n)$ space. Just as with factorial, we can readily formulate an equivalent linear iteration that requires a similar number of steps but constant space.

```
>>> def exp_iter(b, n):
        result = 1
        for _ in range(n):
            result = result * b
        return result
```

We can compute exponentials in fewer steps by using successive squaring. For instance, rather than computing b^8 as

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

we can compute it using three multiplications:

$$\begin{array}{l l} b^2 &= b \cdot b \\ b^4 &= b^2 \cdot b^2 \\ b^8 &= b^4 \cdot b^4 \end{array}$$

This method works fine for exponents that are powers of `2`. We can also take advantage of successive squaring in computing exponentials in general if we use the recursive rule

$$b^n = \left\{\begin{array}{l l} (b^{\frac{1}{2} n})^2 & \mbox{if n is even} \\ b \cdot b^{n-1} & \mbox{if n is odd} \end{array} \right.$$

We can express this method as a recursive function as well:

```
>>> def square(x):
        return x*x
```

```
>>> def fast_exp(b, n):
        if n == 0:
            return 1
        if n % 2 == 0:
            return square(fast_exp(b, n//2))
        else:
            return b * fast_exp(b, n-1)
```

```
>>> fast_exp(2, 100)
1267650600228229401496703205376
```

The process evolved by `fast_exp` grows logarithmically with `n` in both space and number of steps. To see this, observe that computing b^{2n} using `fast_exp` requires only one more multiplication than computing b^n. The size of the exponent we can compute therefore doubles (approximately) with every new multiplication we are allowed. Thus, the number of multiplications required for an exponent of `n` grows about as fast as the

logarithm of `n` base `2`. The process has (\Theta(\log n)) growth. The difference between (\Theta(\log n)) growth and (\Theta(n)) growth becomes striking as (n) becomes large. For example, `fast_exp` for `n` of `1000` requires only `14` multiplications instead of `1000`.

3.3 Recursive Data Structures

In Chapter 2, we introduced the notion of a pair as a primitive mechanism for glueing together two objects into one. We showed that a pair can be implemented using a built-in tuple. The *closure* property of pairs indicated that either element of a pair could itself be a pair.

This closure property allowed us to implement the recursive list data abstraction, which served as our first type of sequence. Recursive lists are most naturally manipulated using recursive functions, as their name and structure would suggest. In this section, we discuss functions for creating and manipulating recursive lists and other recursive data structures.

3.3.1 Processing Recursive Lists

Recall that the recursive list abstract data type represented a list as a first element and the rest of the list. We previously implemented recursive lists using functions, but at this point we can re-implement them using a class. Below, the length (`__len__`) and element selection (`__getitem__`) functions are written recursively to demonstrate typical patterns for processing recursive lists.

```
>>> class Rlist(object):
        """A recursive list consisting of a first element and the rest."""
        class EmptyList(object):
            def __len__(self):
                return 0
        empty = EmptyList()
        def __init__(self, first, rest=empty):
            self.first = first
            self.rest = rest
        def __repr__(self):
            args = repr(self.first)
            if self.rest is not Rlist.empty:
                args += ', {0}'.format(repr(self.rest))
            return 'Rlist({0})'.format(args)
        def __len__(self):
            return 1 + len(self.rest)
        def __getitem__(self, i):
            if i == 0:
                return self.first
            return self.rest[i-1]
```

The definitions of `__len__` and `__getitem__` are in fact recursive, although not explicitly so. The built-in Python function `len` looks for a method called `__len__` when applied to a user-defined object argument. Likewise, the subscript operator looks for a method called `__getitem__` . Thus, these definitions will end up calling themselves. Recursive calls on the rest of the list are a ubiquitous pattern in recursive list processing. This class definition of a recursive list interacts properly with Python's built-in sequence and printing operations.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.rest
Rlist(2, Rlist(3))
>>> len(s)
3
>>> s[1]
2
```

Operations that create new lists are particularly straightforward to express using recursion. For example, we can define a function `extend_rlist`, which takes two recursive lists as arguments and combines the elements of both into a new list.

```
>>> def extend_rlist(s1, s2):
        if s1 is Rlist.empty:
            return s2
        return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

```
>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

Likewise, mapping a function over a recursive list exhibits a similar pattern.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))
```

```
>>> map_rlist(s, square)
Rlist(1, Rlist(4, Rlist(9)))
```

Filtering includes an additional conditional statement, but otherwise has a similar recursive structure.

```
>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        rest = filter_rlist(s.rest, fn)
        if fn(s.first):
            return Rlist(s.first, rest)
        return rest
```

```
>>> filter_rlist(s, lambda x: x % 2 == 1)
Rlist(1, Rlist(3))
```

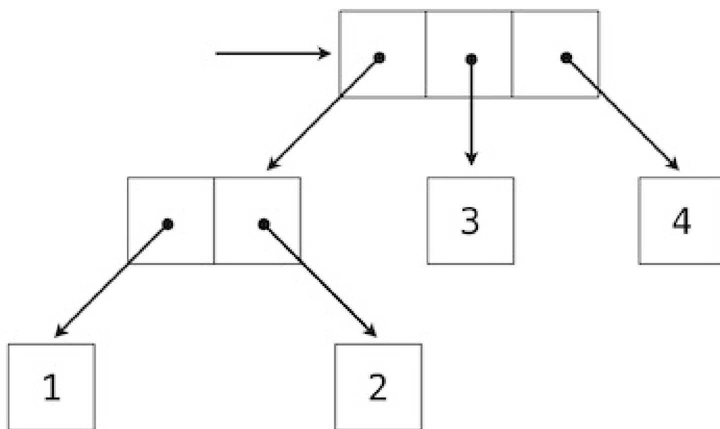
Recursive implementations of list operations do not, in general, require local assignment or `while` statements. Instead, recursive lists are taken apart and constructed incrementally as a consequence of function application. As a result, they have linear orders of growth in both the number of steps and space required.

3.3.2 Hierarchical Structures

Hierarchical structures result from the closure property of data, which asserts for example that tuples can contain other tuples. For instance, consider this nested representation of the numbers 1 through 4.

```
>>> ((1, 2), 3, 4)
((1, 2), 3, 4)
```

This tuple is a length-three sequence, of which the first element is itself a tuple. A box-and-pointer diagram of this nested structure shows that it can also be thought of as a tree with four leaves, each of which is a number.



In a tree, each subtree is itself a tree. As a base condition, any bare element that is not a tuple is itself a simple tree, one with no branches. That is, the numbers are all trees, as is the pair `(1, 2)` and the structure as a whole.

Recursion is a natural tool for dealing with tree structures, since we can often reduce operations on trees to operations on their branches, which reduce in turn to operations on the branches of the branches, and so on, until we reach the leaves of the tree. As an example, we can implement a `count_leaves` function, which returns the total number of leaves of a tree.

```
>>> def count_leaves(tree):
        if type(tree) != tuple:
            return 1
        return sum(map(count_leaves, tree))
```

```
>>> t = ((1, 2), 3, 4)
>>> count_leaves(t)
4
>>> big_tree = ((t, t), 5)
>>> big_tree
((((1, 2), 3, 4), ((1, 2), 3, 4)), 5)
>>> count_leaves(big_tree)
9
```

Just as `map` is a powerful tool for dealing with sequences, mapping and recursion together provide a powerful general form of computation for manipulating trees. For instance, we can square all leaves of a tree using a higher-order recursive function `map_tree` that is structured quite similarly to `count_leaves`.

```
>>> def map_tree(tree, fn):
        if type(tree) != tuple:
            return fn(tree)
        return tuple(map_tree(branch, fn) for branch in tree)
```

```
>>> map_tree(big_tree, square)
((((1, 4), 9, 16), ((1, 4), 9, 16)), 25)
```

Internal values. The trees described above have values only at the leaves. Another common representation of tree-structured data has values for the internal nodes of the tree as well. We can represent such trees using a class.

```
>>> class Tree(object):
        def __init__(self, entry, left=None, right=None):
            self.entry = entry
            self.left = left
            self.right = right
        def __repr__(self):
            args = repr(self.entry)
            if self.left or self.right:
                args += ', {0}, {1}'.format(repr(self.left), repr(self.right))
            return 'Tree({0})'.format(args)
```

The `Tree` class can represent, for instance, the values computed in an expression tree for the recursive implementation of `fib`, the function for computing Fibonacci numbers. The function `fib_tree(n)` below returns a `Tree` that has the nth Fibonacci number as its `entry` and a trace of all previously computed Fibonacci numbers within its branches.

```
>>> def fib_tree(n):
        """Return a Tree that represents a recursive Fibonacci calculation."""
        if n == 1:
            return Tree(0)
        if n == 2:
            return Tree(1)
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        return Tree(left.entry + right.entry, left, right)
```

```
>>> fib_tree(5)
Tree(3, Tree(1, Tree(0), Tree(1)), Tree(2, Tree(1), Tree(1, Tree(0), Tree(1))))
```

This example shows that expression trees can be represented programmatically using tree-structured data. This connection between nested expressions and tree-structured data type plays a central role in our discussion of designing interpreters later in this chapter.

3.3.3 Sets

In addition to the list, tuple, and dictionary, Python has a fourth built-in container type called a `set`. Set literals follow the mathematical notation of elements enclosed in braces. Duplicate elements are removed upon construction. Sets are unordered collections, and so the printed ordering may differ from the element ordering in the set literal.

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
```

Python sets support a variety of operations, including membership tests, length computation, and the standard set operations of union and intersection

```
>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```

In addition to `union` and `intersection`, Python sets support several other methods. The predicates `isdisjoint`, `issubset`, and `issuperset` provide set comparison. Sets are mutable, and can be changed one element at a time using `add`, `remove`, `discard`, and `pop`. Additional methods provide multi-element mutations, such as `clear` and `update`. The Python documentation for sets should be sufficiently intelligible at this point of the course to fill in the details.

Implementing sets. Abstractly, a set is a collection of distinct objects that supports membership testing, union, intersection, and adjunction. Adjoining an element and a set returns a new set that contains all of the original set's elements along with the new element, if it is distinct. Union and intersection return the set of elements that appear in either or both sets, respectively. As with any data abstraction, we are free to implement any functions over any representation of sets that provides this collection of behaviors.

In the remainder of this section, we consider three different methods of implementing sets that vary in their representation. We will characterize the efficiency of these different representations by analyzing the order of growth of set operations. We will use our `Rlist` and `Tree` classes from earlier in this section, which allow for simple and elegant recursive solutions for elementary set operations.

Sets as unordered sequences. One way to represent a set is as a sequence in which no element appears more than once. The empty set is represented by the empty sequence. Membership testing walks recursively through the list.

```
>>> def empty(s):
        return s is Rlist.empty
```

```
>>> def set_contains(s, v):
        """Return True if and only if set s contains v."""
        if empty(s):
            return False
        elif s.first == v:
            return True
        return set_contains(s.rest, v)
```

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> set_contains(s, 2)
True
>>> set_contains(s, 5)
False
```

This implementation of `set_contains` requires (\Theta(n)) time to test membership of an element, where (n) is the size of the set `s`. Using this linear-time function for membership, we can adjoin an element to a set, also in linear time.

```
>>> def adjoin_set(s, v):
        """Return a set containing all elements of s and element v."""
        if set_contains(s, v):
            return s
        return Rlist(v, s)
```

```
>>> t = adjoin_set(s, 4)
>>> t
Rlist(4, Rlist(1, Rlist(2, Rlist(3))))
```

In designing a representation, one of the issues with which we should be concerned is efficiency. Intersecting two sets `set1` and `set2` also requires membership testing, but this time each element of `set1` must be tested for membership in `set2`, leading to a quadratic order of growth in the number of steps, (\Theta(n^2)), for two sets of size (n).

```
>>> def intersect_set(set1, set2):
        """Return a set containing all elements common to set1 and set2."""
        return filter_rlist(set1, lambda v: set_contains(set2, v))
```

```
>>> intersect_set(t, map_rlist(s, square))
Rlist(4, Rlist(1))
```

When computing the union of two sets, we must be careful not to include any element twice. The `union_set` function also requires a linear number of membership tests, creating a process that also includes (\Theta(n^2)) steps.

```
>>> def union_set(set1, set2):
        """Return a set containing all elements either in set1 or set2."""
        set1_not_set2 = filter_rlist(set1, lambda v: not set_contains(set2, v))
        return extend_rlist(set1_not_set2, set2)
```

```
>>> union_set(t, s)
Rlist(4, Rlist(1, Rlist(2, Rlist(3))))
```

Sets as ordered tuples. One way to speed up our set operations is to change the representation so that the set elements are listed in increasing order. To do this, we need some way to compare two objects so that we can say which is bigger. In Python, many different types of objects can be compared using < and > operators, but we will concentrate on numbers in this example. We will represent a set of numbers by listing its elements in increasing order.

One advantage of ordering shows up in set_contains: In checking for the presence of an object, we no longer have to scan the entire set. If we reach a set element that is larger than the item we are looking for, then we know that the item is not in the set:

```
>>> def set_contains(s, v):
        if empty(s) or s.first > v:
            return False
        elif s.first == v:
            return True
        return set_contains(s.rest, v)
```

```
>>> set_contains(s, 0)
False
```

How many steps does this save? In the worst case, the item we are looking for may be the largest one in the set, so the number of steps is the same as for the unordered representation. On the other hand, if we search for items of many different sizes we can expect that sometimes we will be able to stop searching at a point near the beginning of the list and that other times we will still need to examine most of the list. On average we should expect to have to examine about half of the items in the set. Thus, the average number of steps required will be about (\frac{n}{2}). This is still (\Theta(n)) growth, but it does save us, on average, a factor of 2 in the number of steps over the previous implementation.

We can obtain a more impressive speedup by re-implementing intersect_set. In the unordered representation, this operation required (\Theta(n^2)) steps because we performed a complete scan of set2 for each element of set1. But with the ordered representation, we can use a more clever method. We iterate through both sets simultaneously, tracking an element e1 in set1 and e2 in set2. When e1 and e2 are equal, we include that element in the intersection.

Suppose, however, that `e1` is less than `e2`. Since `e2` is smaller than the remaining elements of `set2`, we can immediately conclude that `e1` cannot appear anywhere in the remainder of `set2` and hence is not in the intersection. Thus, we no longer need to consider `e1`; we discard it and proceed to the next element of `set1`. Similar logic advances through the elements of `set2` when `e2 < e1`. Here is the function:

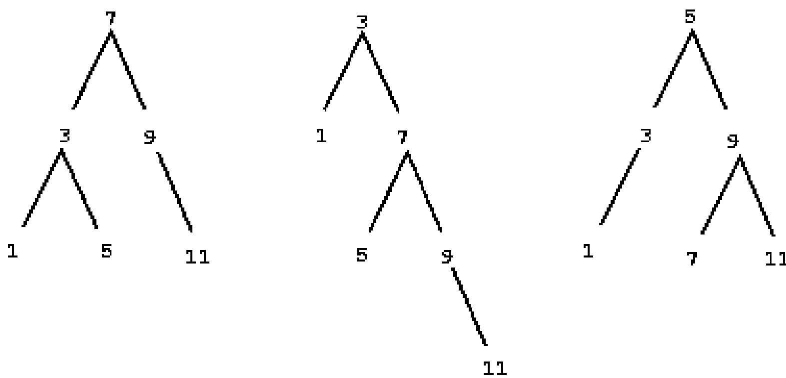
```
>>> def intersect_set(set1, set2):
        if empty(set1) or empty(set2):
            return Rlist.empty
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
        elif e2 < e1:
            return intersect_set(set1, set2.rest)
```

```
>>> intersect_set(s, s.rest)
Rlist(2, Rlist(3))
```

To estimate the number of steps required by this process, observe that in each step we shrink the size of at least one of the sets. Thus, the number of steps required is at most the sum of the sizes of `set1` and `set2`, rather than the product of the sizes, as with the unordered representation. This is (\Theta(n)) growth rather than (\Theta(n^2)) -- a considerable speedup, even for sets of moderate size. For example, the intersection of two sets of size `100` will take around `200` steps, rather than `10,000` for the unordered representation.

Adjunction and union for sets represented as ordered sequences can also be computed in linear time. These implementations are left as an exercise.

Sets as binary trees. We can do better than the ordered-list representation by arranging the set elements in the form of a tree. We use the `Tree` class introduced previously. The `entry` of the root of the tree holds one element of the set. The entries within the `left` branch include all elements smaller than the one at the root. Entries in the `right` branch include all elements greater than the one at the root. The figure below shows some trees that represent the set `{1, 3, 5, 7, 9, 11}`. The same set may be represented by a tree in a number of different ways. The only thing we require for a valid representation is that all elements in the `left` subtree be smaller than the tree `entry` and that all elements in the `right` subtree be larger.



The advantage of the tree representation is this: Suppose we want to check whether a value `v` is contained in a set. We begin by comparing `v` with `entry`. If `v` is less than this, we know that we need only search the `left` subtree; if `v` is greater, we need only search the `right` subtree. Now, if the tree is "balanced," each of these subtrees will be about half the size of the original. Thus, in one step we have reduced the problem of searching a tree of size (n) to searching a tree of size $(\frac{n}{2})$. Since the size of the tree is halved at each step, we should expect that the number of steps needed to search a tree grows as $(\Theta(\log n))$. For large sets, this will be a significant speedup over the previous representations. This `set_contains` function exploits the ordering structure of the tree-structured set.

```
>>> def set_contains(s, v):
        if s is None:
            return False
        elif s.entry == v:
            return True
        elif s.entry < v:
            return set_contains(s.right, v)
        elif s.entry > v:
            return set_contains(s.left, v)
```

Adjoining an item to a set is implemented similarly and also requires $(\Theta(\log n))$ steps. To adjoin a value `v`, we compare `v` with `entry` to determine whether `v` should be added to the `right` or to the `left` branch, and having adjoined `v` to the appropriate branch we piece this newly constructed branch together with the original `entry` and the other branch. If `v` is equal to the `entry`, we just return the node. If we are asked to adjoin `v` to an empty tree, we generate a `Tree` that has `v` as the `entry` and empty `right` and `left` branches. Here is the function:

```
>>> def adjoin_set(s, v):
        if s is None:
            return Tree(v)
        if s.entry == v:
            return s
        if s.entry < v:
            return Tree(s.entry, s.left, adjoin_set(s.right, v))
        if s.entry > v:
            return Tree(s.entry, adjoin_set(s.left, v), s.right)
```

```
>>> adjoin_set(adjoin_set(adjoin_set(None, 2), 3), 1)
Tree(2, Tree(1), Tree(3))
```

Our claim that searching the tree can be performed in a logarithmic number of steps rests on the assumption that the tree is "balanced," i.e., that the left and the right subtree of every tree have approximately the same number of elements, so that each subtree contains about half the elements of its parent. But how can we be certain that the trees we construct will be balanced? Even if we start with a balanced tree, adding elements with `adjoin_set` may produce an unbalanced result. Since the position of a newly adjoined element depends on how the element compares with the items already in the set, we can expect that if we add elements "randomly" the tree will tend to be balanced on the average.

But this is not a guarantee. For example, if we start with an empty set and adjoin the numbers 1 through 7 in sequence we end up with a highly unbalanced tree in which all the left subtrees are empty, so it has no advantage over a simple ordered list. One way to solve this problem is to define an operation that transforms an arbitrary tree into a balanced tree with the same elements. We can perform this transformation after every few `adjoin_set` operations to keep our set in balance.

Intersection and union operations can be performed on tree-structured sets in linear time by converting them to ordered lists and back. The details are left as an exercise.

Python set implementation. The `set` type that is built into Python does not use any of these representations internally. Instead, Python uses a representation that gives constant-time membership tests and adjoin operations based on a technique called *hashing*, which is a topic for another course. Built-in Python sets cannot contain mutable data types, such as lists, dictionaries, or other sets. To allow for nested sets, Python also includes a built-in immutable `frozenset` class that shares methods with the `set` class but excludes mutation methods and operators.

3.4 Exceptions

Programmers must be always mindful of possible errors that may arise in their programs. Examples abound: a function may not receive arguments that it is designed to accept, a necessary resource may be missing, or a connection across a network may be lost. When designing a program, one must anticipate the exceptional circumstances that may arise and take appropriate measures to handle them.

There is no single correct approach to handling errors in a program. Programs designed to provide some persistent service like a web server should be robust to errors, logging them for later consideration but continuing to service new requests as long as possible. On the other hand, the Python interpreter handles errors by terminating immediately and printing an error message, so that programmers can address issues as soon as they arise. In any case, programmers must make conscious choices about how their programs should react to exceptional conditions.

Exceptions, the topic of this section, provides a general mechanism for adding error-handling logic to programs. *Raising an exception* is a technique for interrupting the normal flow of execution in a program, signaling that some exceptional circumstance has arisen, and returning directly to an enclosing part of the program that was designated to react to that circumstance. The Python interpreter raises an exception each time it detects an error in an expression or statement. Users can also raise exceptions with `raise` and `assert` statements.

Raising exceptions. An exception is a object instance with a class that inherits, either directly or indirectly, from the `BaseException` class. The `assert` statement introduced in Chapter 1 raises an exception with the class `AssertionError`. In general, any exception instance can be raised with the `raise` statement. The general form of raise statements are described in the Python docs. The most common use of `raise` constructs an exception instance and raises it.

```
>>> raise Exception('An error occurred')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: an error occurred
```

When an exception is raised, no further statements in the current block of code are executed. Unless the exception is *handled* (described below), the interpreter will return directly to the interactive read-eval-print loop, or terminate entirely if Python was started with a file argument. In addition, the interpreter will print a *stack backtrace*, which is a structured block of text that describes the nested set of active function calls in the branch of execution

in which the exception was raised. In the example above, the file name `<stdin>` indicates that the exception was raised by the user in an interactive session, rather than from code in a file.

Handling exceptions. An exception can be handled by an enclosing `try` statement. A `try` statement consists of multiple clauses; the first begins with `try` and the rest begin with `except` :

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

The `<try suite>` is always executed immediately when the `try` statement is executed. Suites of the `except` clauses are only executed when an exception is raised during the course of executing the `<try suite>` . Each `except` clause specifies the particular class of exception to handle. For instance, if the `<exception class>` is `AssertionError` , then any instance of a class inheriting from `AssertionError` that is raised during the course of executing the `<try suite>` will be handled by the following `<except suite>` . Within the `<except suite>` , the identifier `<name>` is bound to the exception object that was raised, but this binding does not persist beyond the `<except suite>` .

For example, we can handle a `ZeroDivisionError` exception using a `try` statement that binds the name `x` to `0` when the exception is raised.

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0
handling a <class 'ZeroDivisionError'>
>>> x
0
```

A `try` statement will handle exceptions that occur within the body of a function that is applied (either directly or indirectly) within the `<try suite>` . When an exception is raised, control jumps directly to the body of the `<except suite>` of the most recent `try` statement that handles that type of exception.

```
>>> def invert(x):
        result = 1/x  # Raises a ZeroDivisionError if x is 0
        print('Never printed if x is 0')
        return result
```

```
>>> def invert_safe(x):
        try:
            return invert(x)
        except ZeroDivisionError as e:
            return str(e)
```

```
>>> invert_safe(2)
Never printed if x is 0
0.5
>>> invert_safe(0)
'division by zero'
```

This example illustrates that the `print` expression in `invert` is never evaluated, and instead control is transferred to the suite of the `except` clause in `handler`. Coercing the `ZeroDivisionError` `e` to a string gives the human-interpretable string returned by `handler`: `'division by zero'`.

3.4.1 Exception Objects

Exception objects themselves carry attributes, such as the error message stated in an `assert` statement and information about where in the course of execution the exception was raised. User-defined exception classes can carry additional attributes.

In Chapter 1, we implemented Newton's method to find the zeroes of arbitrary functions. The following example defines an exception class that returns the best guess discovered in the course of iterative improvement whenever a `ValueError` occurs. A math domain error (a type of `ValueError`) is raised when `sqrt` is applied to a negative number. This exception is handled by raising an `IterImproveError` that stores the most recent guess from Newton's method as an attribute.

First, we define a new class that inherits from `Exception`.

```
>>> class IterImproveError(Exception):
        def __init__(self, last_guess):
            self.last_guess = last_guess
```

Next, we define a version of `IterImprove`, our generic iterative improvement algorithm. This version handles any `ValueError` by raising an `IterImproveError` that stores the most recent guess. As before, `iter_improve` takes as arguments two functions, each of which takes a single numerical argument. The `update` function returns new guesses, while the `done` function returns a boolean indicating that improvement has converged to a correct value.

```
>>> def iter_improve(update, done, guess=1, max_updates=1000):
        k = 0
        try:
            while not done(guess) and k < max_updates:
                guess = update(guess)
                k = k + 1
            return guess
        except ValueError:
            raise IterImproveError(guess)
```

Finally, we define `find_root`, which returns the result of `iter_improve` applied to a Newton update function returned by `newton_update`, which is defined in Chapter 1 and requires no changes for this example. This version of `find_root` handles an `IterImproveError` by returning its last guess.

```
>>> def find_root(f, guess=1):
        def done(x):
            return f(x) == 0
        try:
            return iter_improve(newton_update(f), done, guess)
        except IterImproveError as e:
            return e.last_guess
```

Consider applying `find_root` to find the zero of the function (2x^2 + \sqrt{x}). This function has a zero at `0`, but evaluating it on any negative number will raise a `ValueError`. Our Chapter 1 implementation of Newton's Method would raise that error and fail to return any guess of the zero. Our revised implementation returns the last guess found before the error.

```
>>> from math import sqrt
>>> find_root(lambda x: 2*x*x + sqrt(x))
-0.030211203830201594
```

While this approximation is still far from the correct answer of `0`, some applications would prefer this coarse approximation to a `ValueError`.

Exceptions are another technique that help us as programs to separate the concerns of our program into modular parts. In this example, Python's exception mechanism allowed us to separate the logic for iterative improvement, which appears unchanged in the suite of the `try` clause, from the logic for handling errors, which appears in `except` clauses. We will also find that exceptions are a very useful feature when implementing interpreters in Python.

3.5 Interpreters for Languages with Combination

The software running on any modern computer is written in a variety of programming languages. There are physical languages, such as the machine languages for particular computers. These languages are concerned with the representation of data and control in terms of individual bits of storage and primitive machine instructions. The machine-language programmer is concerned with using the given hardware to erect systems and utilities for the efficient implementation of resource-limited computations. High-level languages, erected on a machine-language substrate, hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. These languages have means of combination and abstraction, such as procedure definition, that are appropriate to the larger-scale organization of software systems.

Metalinguistic abstraction -- establishing new languages -- plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing interpreters. An interpreter for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression.

We now embark on a tour of the technology by which languages are established in terms of other languages. We will first define an interpreter for a limited language called Calculator that shares the syntax of Python call expressions. We will then develop a sketch interpreters for the Scheme and Logo languages, which is are dialects of Lisp, the second oldest language still in widespread use today. The interpreter we create will be complete in the sense that it will allow us to write fully general programs in Logo. To do so, it will implement the environment model of evaluation that we have developed over the course of this text.

3.5.1 Calculator

Our first new language is Calculator, an expression language for the arithmetic operations of addition, subtraction, multiplication, and division. Calculator shares Python's call expression syntax, but its operators are more flexible in the number of arguments they accept. For instance, the Calculator operators `add` and `mul` take an arbitrary number of arguments:

```
calc> add(1, 2, 3, 4)
10
calc> mul()
1
```

The `sub` operator has two behaviors. With one argument, it negates the argument. With at least two arguments, it subtracts all but the first from the first. The `div` operator has the semantics of Python's `operator.truediv` function and takes exactly two arguments:

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

As in Python, call expression nesting provides a means of combination in the Calculator language. To condense notation, the names of operators can also be replaced by their standard symbols:

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

We will implement an interpreter for Calculator in Python. That is, we will write a Python program that takes a string as input and either returns the result of evaluating that string if it is a well-formed Calculator expression or raises an appropriate exception if it is not. The core of the interpreter for the Calculator language is a recursive function called `calc_eval` that evaluates a tree-structured expression object.

Expression trees. Until this point in the course, expression trees have been conceptual entities to which we have referred in describing the process of evaluation; we have never before explicitly represented expression trees as data in our programs. In order to write an interpreter, we must operate on expressions as data. In the course of this chapter, many of the concepts introduced in previous chapters will finally by realized in code.

A primitive expression is just a number in Calculator, either an `int` or `float` type. All combined expressions are call expressions. A call expression is represented as a class `Exp` that has two attribute instances. The `operator` in Calculator is always a string: an arithmetic operator name or symbol. The `operands` are either primitive expressions or themselves instances of `Exp`.

```
>>> class Exp(object):
        """A call expression in Calculator."""
        def __init__(self, operator, operands):
            self.operator = operator
            self.operands = operands
        def __repr__(self):
            return 'Exp({0}, {1})'.format(repr(self.operator), repr(self.operands))
        def __str__(self):
            operand_strs = ', '.join(map(str, self.operands))
            return '{0}({1})'.format(self.operator, operand_strs)
```

An `Exp` instance defines two string methods. The `__repr__` method returns Python expression, while the `__str__` method returns a Calculator expression.

```
>>> Exp('add', [1, 2])
Exp('add', [1, 2])
>>> str(Exp('add', [1, 2]))
'add(1, 2)'
>>> Exp('add', [1, Exp('mul', [2, 3, 4])])
Exp('add', [1, Exp('mul', [2, 3, 4])])
>>> str(Exp('add', [1, Exp('mul', [2, 3, 4])]))
'add(1, mul(2, 3, 4))'
```

This final example demonstrates how the `Exp` class represents the hierarchical structure in expression trees by including instances of `Exp` as elements of `operands`.

Evaluation. The `calc_eval` function itself takes an expression as an argument and returns its value. It classifies the expression by its form and directs its evaluation. For Calculator, the only two syntactic forms of expressions are numbers and call expressions, which are `Exp` instances. Numbers are *self-evaluating*; they can be returned directly from `calc_eval`. Call expressions require function application.

```
>>> def calc_eval(exp):
        """Evaluate a Calculator expression."""
        if type(exp) in (int, float):
            return exp
        elif type(exp) == Exp:
            arguments = list(map(calc_eval, exp.operands))
            return calc_apply(exp.operator, arguments)
```

Call expressions are evaluated by first recursively mapping the `calc_eval` function to the list of operands to compute a list of `arguments`. Then, the operator is applied to those arguments in a second function, `calc_apply`.

The Calculator language is simple enough that we can easily express the logic of applying each operator in the body of a single function. In `calc_apply`, each conditional clause corresponds to applying one operator.

```
>>> from operator import mul
>>> from functools import reduce
>>> def calc_apply(operator, args):
        """Apply the named operator to a list of args."""
        if operator in ('add', '+'):
            return sum(args)
        if operator in ('sub', '-'):
            if len(args) == 0:
                raise TypeError(operator + ' requires at least 1 argument')
            if len(args) == 1:
                return -args[0]
            return sum(args[:1] + [-arg for arg in args[1:]])
        if operator in ('mul', '*'):
            return reduce(mul, args, 1)
        if operator in ('div', '/'):
            if len(args) != 2:
                raise TypeError(operator + ' requires exactly 2 arguments')
            numer, denom = args
            return numer/denom
```

Above, each suite computes the result of a different operator, or raises an appropriate `TypeError` when the wrong number of arguments is given. The `calc_apply` function can be applied directly, but it must be passed a list of *values* as arguments rather than a list of operand expressions.

```
>>> calc_apply('+', [1, 2, 3])
6
>>> calc_apply('-', [10, 1, 2, 3])
4
>>> calc_apply('*', [])
1
>>> calc_apply('/', [40, 5])
8.0
```

The role of `calc_eval` is to make proper calls to `calc_apply` by first computing the value of operand sub-expressions before passing them as arguments to `calc_apply`. Thus, `calc_eval` can accept a nested expression.

```
>>> e = Exp('add', [2, Exp('mul', [4, 6])])
>>> str(e)
'add(2, mul(4, 6))'
>>> calc_eval(e)
26
```

The structure of `calc_eval` is an example of dispatching on type: the form of the expression. The first form of expression is a number, which requires no additional evaluation step. In general, primitive expressions that do not require an additional evaluation step are called *self-evaluating*. The only self-evaluating expressions in our Calculator language are numbers, but a general programming language might also include strings, boolean values, etc.

Read-eval-print loops. A typical approach to interacting with an interpreter is through a read-eval-print loop, or REPL, which is a mode of interaction that reads an expression, evaluates it, and prints the result for the user. The Python interactive session is an example of such a loop.

An implementation of a REPL can be largely independent of the interpreter it uses. The function `read_eval_print_loop` below takes as input a line of text from the user with the built-in `input` function. It constructs an expression tree using the language-specific `calc_parse` function, defined in the following section on parsing. Finally, it prints the result of applying `calc_eval` to the expression tree returned by `calc_parse`.

```
>>> def read_eval_print_loop():
        """Run a read-eval-print loop for calculator."""
        while True:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
```

This version of `read_eval_print_loop` contains all of the essential components of an interactive interface. An example session would look like:

```
calc> mul(1, 2, 3)
6
calc> add()
0
calc> add(2, div(4, 8))
2.5
```

This loop implementation has no mechanism for termination or error handling. We can improve the interface by reporting errors to the user. We can also allow the user to exit the loop by signalling a keyboard interrupt (`Control-C` on UNIX) or end-of-file exception (`Control-D` on UNIX). To enable these improvements, we place the original suite of the `while` statement within a `try` statement. The first `except` clause handles `SyntaxError` exceptions raised by `calc_parse` as well as `TypeError` and `ZeroDivisionError` exceptions raised by `calc_eval`.

```
>>> def read_eval_print_loop():
        """Run a read-eval-print loop for calculator."""
        while True:
            try:
                expression_tree = calc_parse(input('calc> '))
                print(calc_eval(expression_tree))
            except (SyntaxError, TypeError, ZeroDivisionError) as err:
                print(type(err).__name__ + ':', err)
            except (KeyboardInterrupt, EOFError):  # <Control>-D, etc.
                print('Calculation completed.')
                return
```

This loop implementation reports errors without exiting the loop. Rather than exiting the program on an error, restarting the loop after an error message lets users revise their expressions. Upon importing the `readline` module, users can even recall their previous inputs using the up arrow or `Control-P`. The final result provides an informative error reporting interface:

```
calc> add
SyntaxError: expected ( after add
calc> div(5)
TypeError: div requires exactly 2 arguments
calc> div(1, 0)
ZeroDivisionError: division by zero
calc> ^DCalculation completed.
```

As we generalize our interpreter to new languages other than Calculator, we will see that the `read_eval_print_loop` is parameterized by a parse function, an evaluation function, and the exception types handled by the `try` statement. Beyond these changes, all REPLs can be implemented using the same structure.

3.5.2 Parsing

Parsing is the process of generating expression trees from raw text input. It is the job of the evaluation function to interpret those expression trees, but the parser must supply well-formed expression trees to the evaluator. A parser is in fact a composition of two components: a lexical analyzer and a syntactic analyzer. First, the lexical analyzer partitions the input string into *tokens*, which are the minimal syntactic units of the language, such as names and symbols. Second, the syntactic analyzer constructs an expression tree from this sequence of tokens.

```
>>> def calc_parse(line):
        """Parse a line of calculator input and return an expression tree."""
        tokens = tokenize(line)
        expression_tree = analyze(tokens)
        if len(tokens) > 0:
            raise SyntaxError('Extra token(s): ' + ' '.join(tokens))
        return expression_tree
```

The sequence of tokens produced by the lexical analyzer, called `tokenize`, is consumed by the syntactic analyzer, called `analyze`. In this case, we define `calc_parse` to expect only one well-formed Calculator expression. Parsers for some languages are designed to accept multiple expressions delimited by new line characters, semicolons, or even spaces. We defer this additional complexity until we introduce the Logo language below.

Lexical analysis. The component that interprets a string as a token sequence is called a *tokenizer* or *lexical analyzer*. In our implementation, the tokenizer is a function called `tokenize`. The Calculator language consists of symbols that include numbers, operator

names, and operator symbols, such as `+`. These symbols are always separated by two types of delimiters: commas and parentheses. Each symbol is its own token, as is each comma and parenthesis. Tokens can be separated by adding spaces to the input string and then splitting the string at each space.

```
>>> def tokenize(line):
        """Convert a string into a list of tokens."""
        spaced = line.replace('(',' ( ').replace(')',' ) ').replace(',', ' , ')
        return spaced.split()
```

Tokenizing a well-formed Calculator expression keeps names intact, but separates all symbols and delimiters.

```
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']
```

Languages with a more complicated syntax may require a more sophisticated tokenizer. In particular, many tokenizers resolve the syntactic type of each token returned. For example, the type of a token in Calculator may be an operator, a name, a number, or a delimiter. This classification can simplify the process of *parsing* the token sequence.

Syntactic analysis. The component that interprets a token sequence as an expression tree is called a *syntactic analyzer*. In our implementation, syntactic analysis is performed by a recursive function called `analyze`. It is recursive because analyzing a sequence of tokens often involves analyzing a subsequence of those tokens into an expression tree, which itself serves as a branch (i.e., operand) of a larger expression tree. Recursion generates the hierarchical structures consumed by the evaluator.

The `analyze` function expects a list of tokens that begins with a well-formed expression. It analyzes the first token, coercing strings that represent numbers into numeric values. It then must consider the two legal expression types in the Calculator language. Numeric tokens are themselves complete, primitive expression trees. Combined expressions begin with an operator and follow with a list of operand expressions delimited by parentheses. Operands are analyzed by the `analyze_operands` function, which recursively calls `analyze` on each operand expression. We begin with an implementation that does not check for syntax errors.

```
>>> def analyze(tokens):
        """Create a tree of nested lists from a sequence of tokens."""
        token = analyze_token(tokens.pop(0))
        if type(token) in (int, float):
            return token
        else:
            tokens.pop(0)  # Remove (
            return Exp(token, analyze_operands(tokens))
```

```
>>> def analyze_operands(tokens):
        """Read a list of comma-separated operands."""
        operands = []
        while tokens[0] != ')':
            if operands:
                tokens.pop(0)  # Remove ,
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
        return operands
```

Finally, we need to implement `analyze_token`. The `analyze_token` function that converts number literals into numbers. Rather than implementing this logic ourselves, we rely on built-in Python type coercion, using the `int` and `float` constructors to convert tokens to those types.

```
>>> def analyze_token(token):
        """Return the value of token if it can be analyzed as a number, or token."""
        try:
            return int(token)
        except (TypeError, ValueError):
            try:
                return float(token)
            except (TypeError, ValueError):
                return token
```

Our implementation of `analyze` is complete; it correctly parses well-formed Calculator expressions into expression trees. These trees can be converted back into Calculator expressions by the `str` function.

```
>>> expression = 'add(2, mul(4, 6))'
>>> analyze(tokenize(expression))
Exp('add', [2, Exp('mul', [4, 6])])
>>> str(analyze(tokenize(expression)))
'add(2, mul(4, 6))'
```

The `analyze` function is meant to return only well-formed expression trees, and so it must detect errors in the syntax of its input. In particular, it must detect that expressions are complete, correctly delimited, and use only known operators. The following revisions ensure that each step of the syntactic analysis finds the token it expects.

```
>>> known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']
```

```
>>> def analyze(tokens):
        """Create a tree of nested lists from a sequence of tokens."""
        assert_non_empty(tokens)
        token = analyze_token(tokens.pop(0))
        if type(token) in (int, float):
            return token
        if token in known_operators:
            if len(tokens) == 0 or tokens.pop(0) != '(':
                raise SyntaxError('expected ( after ' + token)
            return Exp(token, analyze_operands(tokens))
        else:
            raise SyntaxError('unexpected ' + token)
```

```
>>> def analyze_operands(tokens):
        """Analyze a sequence of comma-separated operands."""
        assert_non_empty(tokens)
        operands = []
        while tokens[0] != ')':
            if operands and tokens.pop(0) != ',':
                raise SyntaxError('expected ,')
            operands.append(analyze(tokens))
            assert_non_empty(tokens)
        tokens.pop(0)  # Remove )
        return elements
```

```
>>> def assert_non_empty(tokens):
        """Raise an exception if tokens is empty."""
        if len(tokens) == 0:
            raise SyntaxError('unexpected end of line')
```

Informative syntax errors improve the usability of an interpreter substantially. Above, the `SyntaxError` exceptions that are raised include a description of the problem encountered. These error strings also serve to document the definitions of these analysis functions.

This definition completes our Calculator interpreter. A single Python 3 source file calc.py is available for your experimentation. Our interpreter is robust to errors, in the sense that every input that a user enters at the `calc>` prompt will either be evaluated to a number or raise an appropriate error that describes why the input is not a well-formed Calculator expression.

3.6 Interpreters for Languages with Abstraction

The Calculator language provides a means of combination through nested call expressions. However, there is no way to define new operators, give names to values, or express general methods of computation. In summary, Calculator does not support abstraction in any way. As a result, it is not a particularly powerful or general programming language. We now turn to the task of defining a general programming language that supports abstraction by binding names to values and defining new operations.

Rather than extend our simple Calculator language further, we will begin anew and develop an interpreter for the Logo language. Logo is not a language invented for this course, but instead a classic instructional language with dozens of interpreter implementations and its own developer community.

Unlike the previous section, which presented a complete interpreter as Python source code, this section takes a descriptive approach. The companion project asks you to implement the ideas presented here by building a fully functional Logo interpreter.

3.6.1 The Scheme Language

Scheme is a dialect of Lisp, the second-oldest programming language that is still widely used today (after Fortran). Scheme was first described in 1975 by Gerald Sussman and Guy Steele. From the introduction to the `Revised(4) Report on the Algorithmic Language Scheme` _,

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

We refer you to this Report for full details of the Scheme language. We'll touch on highlights here. We've used examples from the Report in the descriptions below..

Despite its simplicity, Scheme is a real programming language and in many ways is similar to Python, but with a minimum of "syntactic sugar"[1]. Basically, *all* operations take the form of function calls. Here, we will describe a representative subset of the full Scheme language described in the report.

| [1] | Regrettably, this has become less true in more recent revisions of the Scheme language, such as the *Revised(6) Report*, so here, we'll stick with previous versions. |

There are several implementations of Scheme available, which add on various additional procedures. At Berkeley, we've used a modified version of the Stk interpreter, which is also available as `stk` on our instructional servers. Unfortunately, it is not particularly conformant to the official specification, but it will do for our purposes.

Using the Interpreter. As with the Python interpreter[#], expressions typed to the Stk interpreter are evaluated and printed by what is known as a *read-eval-print loop*:

```
>>> 3
3
>>> (- (/ (* (+ 3 7 10) (- 1000 8)) 992) 17)
3
>>> (define (fib n) (if (< n 2) n (+ (fib (- n 2)) (fib (- n 1)))))
fib
>>> '(1 (7 19))
(1 (7 19))
```

| [2] | In our examples, we use the same notation as for Python: `>>>` and `...` to indicate lines input to the interpreter and unprefixed lines to indicate output. In reality, Scheme interpreters use different prompts. STk, for example, prompts with `STk>` and does not prompt for continuation lines. The Python conventions, however, make it clearer what is input and what is output. |

Values in Scheme. Values in Scheme generally have their counterparts in Python.

> Booleans > > The values true and false, denoted `#t` and `#f`. In Scheme, the only false value (in the Python sense) is `#f`. > > Numbers > > These include integers of arbitrary precision, rational numbers, complex numbers, and "inexact" (generally floating-point) numbers. Integers may be denoted either in standard decimal notation or in other radixes by prefixing a numeral with `#o` (octal), `#x` (hexadecimal), or `#b` (binary). > > Symbols > > Symbols are a kind of string, but are denoted without quotation marks. The valid characters include letters, digits, and: > > `> ! $ % & * / : < = > ? ^ _ ~ + - . @ > >` > > When input by the `read` function, which reads Scheme expressions (and which the interpreter uses to input program text), upper and lower case characters in symbols are not distinguished (in the STk implementation, converted to lower case). Two symbols with the same denotation denote the same object (not just two objects that happen to have the same contents). > > Pairs and Lists > > A pair is an object containing two components (of any types), called its `car` and `cdr`. A pair whose `car` is `A` and whose `cdr` is `B` is denoted `(A . B)`. Pairs (like tuples in Python) can represent lists, trees, and arbitrary hierarchical structures. > > A standard Scheme list consists either of the special empty list value (denoted ()), or of a pair that contains the first item of the list as its `car` and the rest of the list as its `cdr`. Thus, the list consisting of the integers 1, 2, and 3 would be represented: > >

`> (1 . (2 . (3 . ()))) > >` > > Lists are so pervasive that Scheme allows one to abbreviate `(a . ())` as `(a)`, and allows one to abbreviate `(a . (b ...))` as `(a b ...)`. Thus, the list above is usually written: > > `> (1 2 3) > >` > > Procedures (functions) > > As in Python, a procedure (or function) value represents some computation that can be invoked by a function call supplying argument values. Procedures may either be primitives, supplied by the Scheme runtime system, or they may be constructed out of Scheme expression(s) and an environment (exactly as in Python). There is no direct denotation for function values, although there are predefined identifiers that are bound to primitive functions and there are Scheme expressions that, when evaluated, produce new procedure values. > > Other Types > > Scheme also supports characters and strings (like Python strings, except that Scheme distinguishes characters from strings), and vectors (like Python lists).

Program Denotations As with other versions of Lisp, Scheme's data values double as representations of programs. For example, the Scheme list:

```
(+ x (* 10 y))
```

can, depending on how it is used, represent either a 3-item list (whose last item is also a 3-item list), or it can represent a Scheme expression for computing (x+10y). To interpret a Scheme value as a program, we consider the type of value, and evaluate as follows:

> *Integers, booleans, characters, strings, and vectors evaluate to themselves. Thus, the expression* `5` *evaluates to 5.* > Bare symbols serve as variables. Their values are determined by the current environment in which they are being evaluated, just as in Python. > *Non-empty lists are interpreted in two different ways, depending on their first component:* > If the first component is one of the symbols denoting a *special form*, described below, the evaluation proceeds by the rules for that special form. > *In all other cases (called combinations), the items in the list are evaluated (recursively) in some unspecified order. The value of the first item must be a function value. That value is called, with the values of the remaining items in the list supplying the arguments.* > Other Scheme values (in particular, pairs that are not lists) are erroneous as programs.

For example:

```
>>> 5              ; A literal.
5
>>> (define x 3)   ; A special form that creates a binding for symbol
x                   ; x.
>>> (+ 3 (* 10 x)) ; A combination.  Symbol + is bound to the primitive
33                  ; add function and * to primitive multiply.
```

Primitive Special Forms. The special forms denote things such as control structures, function definitions, or class definitions in Python: constructs in which the operands are not simply evaluated immediately, as they are in calls.

First, a couple of common constructs used in the forms:

> *EXPR-SEQ* > > Simply a sequence of expressions, such as: > > `> (+ 3 2) x (* y z) > >` > > When this appears in the definitions below, it refers to a sequence of expressions that are evaluated from left to right, with the value of the sequence (if needed) being the value of the last expression. > > *BODY* > > Several constructs have "bodies", which are *EXPR-SEQ_s, as above, optionally preceded by one or more Definitions. Their value is that of their _EXPR-SEQ*. See the section on Internal Definitions for the interpretation of these definitions.

Here is a representative subset of the special forms:

> Definitions > > Definitions may appear either at the top level of a program (that is, not enclosed in another construct). > > > (define *SYM EXPR*) > > > > This evaluates *EXPR* and binds its value to the symbol *SYM* in the current environment. > > > > (define (*SYM ARGUMENTS*) *BODY*) > > > > This is equivalent to > > > > > (define *SYM* (lambda (*ARGUMENTS*) *BODY*)) > > (lambda (*ARGUMENTS*) *BODY*) > > This evaluates to a function. *ARGUMENTS* is usually a list (possibly empty) of distinct symbols that gives names to the arguments of the function, and indicates their number. It is also possible for *ARGUMENTS* to have the form: > > `> (sym1 sym2 ... symn . symr) > >` > > (that is, instead of ending in the empty list like a normal list, the last `cdr` is a symbol). In this case, `symr` will be bound to the list of trailing argument values (argument n+1 onward). > > When the resulting function is called, *ARGUMENTS* are bound to the argument values in a fresh environment frame that extends the environment in which the `lambda` expression was evaluated (just like Python). Then the *BODY* is evaluated and its value returned as the value of the call. > > (if *COND-EXPR TRUE-EXPR OPTIONAL-FALSE-EXPR*) > > Evaluates *COND-EXPR*, and if its value is not `#f`, then evaluates *TRUE-EXPR*, and the result is the value of the `if`. If *COND-EXPR* evaluates to `#f` and *OPTIONAL-FALSE-EXPR* is present, it is evaluated and its result is the value of the `if`. If it is absent, the value of the `if` is unspecified. > > (set! *SYMBOL EXPR*) > > Evaluates *EXPR* and replaces the binding of *SYMBOL* with the resulting value. *SYMBOL* must be bound, or there is an error. In contrast to Python's default, this replaces the binding of *SYMBOL* in the first enclosing environment frame that defines it, which is not always the innermost frame. > > (quote *EXPR*) or '*EXPR* > > One problem with using Scheme data structures as program representations is that one needs a way to indicate when a particular symbol or list represents literal data to be manipulated by a program, and when it is program text that is intended to be evaluated. The `quote` form evaluates to *EXPR* itself, without further

evaluating *EXPR*. (The alternative form, with leading apostrophe, gets converted to the first form by Scheme's expression reader.) For example: > >

```
&gt; &gt;&gt;&gt; (+ 1 2) &gt; 3 &gt; &gt;&gt;&gt; '(+ 1 2) &gt; (+ 1 2) &gt; &gt;&gt;&gt;
```

Derived Special Forms

A *derived construct* is one that can be translated into primitive constructs. Their purpose is to make programs more concise or clear for the reader. In Scheme, we have

> (begin *EXPR-SEQ*) > > Simply evaluates and yields the value of the *EXPR-SEQ*. This construct is simply a way to execute a sequence of expressions in a context (such as an `if`) that requires a single expression. > > (and *EXPR1 EXPR2* ...) > > Each *EXPR* is evaluated from left to right until one returns `#f` or the *EXPR_s are exhausted. The value is that of the last _EXPR* evaluated, or `#t` if the list of *EXPR_s is empty. For example: > >*

```
&gt; &gt;&gt;&gt; (and (= 2 2) (&gt; 2 1)) &gt; #t &gt; &gt;&gt;&gt; (and (&lt; 2 2) (&gt;
```

> > (or _EXPR1 EXPR2 ...) > > Each *EXPR* is evaluated from left to right until one returns a value other than `#f` or the *EXPR_s are exhausted. The value is that of the last _EXPR* evaluated, or `#f` if the list of *EXPR_s is empty: For example: > >*

```
&gt; &gt;&gt;&gt; (or (= 2 2) (&gt; 2 3)) &gt; #t &gt; &gt;&gt;&gt; (or (= 2 2) '(a b)) &g
```

> > (cond _CLAUSE1 CLAUSE2 ...) > > Each *CLAUSEi* is processed in turn until one succeeds, and its value becomes the value of the `cond`. If no clause succeeds, the value is unspecified. Each clause has one of three possible forms. The form > > > (*TEST-EXPR EXPR-SEQ*) > > succeeds if *TEST-EXPR* evaluates to a value other than `#f`. In that case, it evaluates *EXPR-SEQ* and yields its value. The *EXPR-SEQ* may be omitted, in which case the value is that of *TEST-EXPR* itself. > > The last clause may have the form > > > (else *EXPR-SEQ*) > > which is equivalent to > > > (#t *EXPR-SEQ*) > > Finally, the form > > > (*TEST_EXPR* => *EXPR*) > > succeeds if *TEST_EXPR* evaluates to a value other than `#f`, call it *V*. If it succeeds, the value of the `cond` construct is that returned by (*EXPR V*). That is, *EXPR* must evaluate to a one-argument function, which is applied to the value of *TEST_EXPR*. > > For example: > >

```
&gt; &gt;&gt;&gt; (cond ((&gt; 3 2) 'greater) &gt; ... ((&lt; 3 2) 'less))) &gt; greater &g
```

> > (case *KEY-EXPR CLAUSE1 CLAUSE2* ...) > > Evaluates *KEY-EXPR* to produce a value, *K*. Then matches *K* against each *CLAUSE1* in turn until one succeeds, and returns the value of that clause. If no clause succeeds, the value is unspecified. Each clause has the form > > > ((*DATUM1 DATUM2* ...) *EXPR-SEQ*) > > The *DATUM_s are Scheme values (they are _not* evaluated). The clause succeeds if *K* matches one of the *DATUM* values (as determined by the `eqv?` function described below.) If the clause succeeds, its *EXPR-SEQ* is evaluated and its value becomes the value of the `case`. The last clause may have the form > > > (else *EXPR-SEQ*) > > which always succeeds. For example: > >

```
&gt; &gt;&gt;&gt; (case (* 2 3) &gt; ... ((2 3 5 7) 'prime) &gt; ... ((1 4 6 8 9) 'composi
```

> > (let *BINDINGS BODY*) > > *BINDINGS* is a list of pairs of the form > > > ((*VAR1 INIT1*) (*VAR2 INIT2*) ...) > > where the *VAR_s are (distinct) symbols and the _INIT_s are expressions. This first evaluates the _INIT* expressions, then creates a new frame that binds

those values to the *VAR_s, and then evaluates the _BODY* in that new environment, returning its value. In other words, this is equivalent to the call > > > ((lambda (*VAR1 VAR2* ...) *BODY*)*INIT1 INIT2* ...) > > Thus, any references to the *VAR_s in the _INIT* expressions refers to the definitions (if any) of those symbols *outside* of the `let` construct. For example: > >

```
&gt; &gt;&gt;&gt; (let ((x 2) (y 3)) &gt; ... (* x y)) &gt; 6 &gt; &gt;&gt;&gt; (let ((x 2
```

> > (let *BINDINGS BODY*) > > *The syntax of BINDINGS is the same as for* `let`*. This is equivalent to > > > (let ((VAR1 INIT1))...(let ((VARn INITn))BODY)) > > In other words, it is like* `let` *except that the new binding of VAR1 is visible in subsequent INIT_s as well as in the _BODY, and similarly for VAR2. For example: > > ``` > >>> (define x 3) > x > >>> (define y 4) > y > >>> (let ((x 5) (y (+ x 1))) y) > 4 > >>> (let* ((x 5) (y (+ x 1))) y) > 6 > >

```
&gt; &gt; (letrec _BINDINGS_ _BODY_) &gt; &gt; Again, the syntax is as for `let`. In this
```

> (letrec ((even? > (lambda (n) > (if (zero? n) > #t > (odd? (- n 1))))) > (odd? > (lambda (n) > (if (zero? n) > #f > (even? (- n 1)))))) > (even? 88)) > > ```

Internal Definitions. When a *BODY* begins with a sequence of `define` constructs, they are known as "internal definitions" and are interpreted a little differently from top-level definitions. Specifically, they work like letrec does.

> *First, bindings are created for all the names defined by the* `define` *statements, initially bound to undefined values.* > Then the values are filled in by the defines.

As a result, a sequence of internal function definitions can be mutually recursive, just as `def` statements in Python that are nested inside a function can be:

```
>>> (define (hard-even? x)     ;; An outer-level definition
...      (define (even? n)      ;; Inner definition
...          (if (zero? n)
...              #t
...              (odd? (- n 1))))
...      (define (odd? n)       ;; Inner definition
...          (if (zero? n)
...              #f
...              (even? (- n 1))))
...      (even? x))
>>> (hard-even? 22)
#t
```

Predefined Functions. There is a large collection of predefined functions, all bound to names in the global environment, and we'll simply illustrate a few here; the rest are catalogued in the Revised(4) Scheme Report. Function calls are not "special" in that they all use the same completely uniform evaluation rule: recursively evaluate all items (including the operator), and then apply the operator's value (which must be a function) to the operands' values.

> ***Arithmetic:*** *Scheme provides the standard arithmetic operators, many with familiar denotations, although the operators uniformly appear before the operands: >*

>

>

> *``` > >>> ; Semicolons introduce one-line comments. > >>> ; Compute (3+7+10)*(1000-8) // 992 - 17 > >>> (- (quotient (*(+ 3 7 10) (- 1000 8))) 17) > 3 > >>> (remainder 27 4) > 3 > >>> (- 17) > -17 >*

>

```
&gt; &gt; &gt; &gt; Similarly, there are the usual numeric comparison operators, extended
```

> > >>> *(< 0 5) > > #t > > >>> (>= 100 10 10 0) > > #t > > >>> (= 21 (* 7 3) (+ 19 2)) > > #t > > >>> (not (= 15 14)) > > #t > > >>> (zero? (- 7 7)) > > #t > > > >

```
&gt; &gt; &gt; &gt; `not`, by the way, is a function, not a special form like `and` or `or
```

> >>> (cons 'a 'b) > (a . b) > >>> (list 'a 'b) > (a b) > >>> (cons 'a (cons 'b '())) > (a b) > >>> (car (cons 'a 'b)) > a > >>> (cdr (cons 'a 'b)) > b > >>> (cdr (list a b)) > (b) > >>> (cadr '(a b)) ; An abbreviation for (car (cdr '(a b))) > b > >>> (cddr '(a b)) ; Similarly, an abbreviation for (cdr (cdr '(a b))) > () > >>> (list-tail '(a b c) 0) > (a b c) > >>> (list-tail '(a b c) 1) > (b c) > >>> (list-ref '(a b c) 0) > a > >>> (list-ref '(a b c) 2) > c > >>> (append '(a b) '(c d) '() '(e)) > (a b c d e) > >>> ; All but the last list is copied. The last is shared, so: > >>> (define L1 (list 'a 'b 'c)) > >>> (define L2 (list 'd)) > >>> (define L3 (append L1 L2)) > >>> (set-car! L1 1) > >>> (set-car! L2 2) > >>> L3 > (a b c 2) > >>> (null? '()) > #t > >>> (list? '()) > #t > >>> (list? '(a b)) > #t > >>> (list? '(a . b)) > #f >

>

```
&gt; &gt; &gt; * **Equivalence:** The `=` operation is for numbers. For general equality o
```

> >>> (eqv? 'a 'a) > #t > >>> (eqv? 'a 'b) > #f > >>> (eqv? 100 (+ 50 50)) > #t > >>> (eqv? (list 'a 'b) (list 'a 'b)) > #f > >>> (equal? (list 'a 'b) (list 'a 'b)) > #t >

>

```
&gt; &gt; &gt; * **Types:** Each type of value satisfies exactly one of the basic type pre
```

> >>> (boolean? #f) > #t > >>> (integer? 3) > #t > >>> (pair? '(a b)) > #t > >>> (null? '()) > #t > >>> (symbol? 'a) > #t > >>> (procedure? +) > #t >

>

```
&gt; &gt; &gt; * **Input and Output:** Scheme interpreters typically run a read-eval-print
```

> >>> (begin (display 'a) (display 'b) (newline)) > ab >

>

```
&gt; &gt; &gt; &gt; Thus, `(display x)` is somewhat akin to Python's &gt; &gt; &gt; &gt; &
```

> >>> (read) > >>> (a b c) > (a b c) >

>

```
&gt; &gt; &gt; * **Evaluation:** The `apply` function provides direct access to the functi
```

> > >>> (apply cons '(1 2)) > > (1 . 2) > > >>> ;; Apply the function f to the arguments in L after g is > > >>> ;; applied to each of them > > >>> (define (compose-list f g L) > > ... (apply f (map g L))) > > >>> (compose-list + (lambda (x) (* x x)) '(1 2 3)) > > 14 > > > >

```
&gt; &gt; &gt; &gt; An extension allows for some "fixed" arguments at the beginning: &gt; &
```

> > >>> (apply + 1 2 '(3 4 5)) > > 15 > > > >

```
&gt; &gt; &gt; &gt; The following function is not in [Revised(4) Scheme](http://people.csa
```

> >>> (eval '(+ 1 2)) > 3 >

>

```
&gt; &gt; &gt; &gt; That is, `eval` evaluates a piece of Scheme data that represents a cor
```

> > >>> (define (incr n) (lambda (x) (+ n x))) > > >>> (define add5 (incr 5)) > > >>> (add5 13) > > 18 > > >>> (eval 'n (procedure-environment add5)) > > 5 > > > > ```

3.6.2 The Logo Language

Logo is another dialect of Lisp. It was designed for educational use, and so many design decisions in Logo are meant to make the language more comfortable for a beginner. For example, most Logo procedures are invoked in prefix form (first the procedure name, then the arguments), but the common arithmetic operators are also provided in the customary infix form. The brilliance of Logo is that its simple, approachable syntax still provides amazing expressivity for advanced programmers.

The central idea in Logo that accounts for its expressivity is that its built-in container type, the Logo `sentence` (also called a `list`), can easily store Logo source code! Logo programs can write and interpret Logo expressions as part of their evaluation process. Many dynamic languages support code generation, including Python, but no language makes code generation quite as fun and accessible as Logo.

You may want to download a fully implemented Logo interpreter at this point to experiment with the language. The standard implementation is Berkeley Logo (also known as UCBLogo), developed by Brian Harvey and his Berkeley students. For macintosh uses, ACSLogo is compatible with the latest version of Mac OSX and comes with a user guide that introduces many features of the Logo language.

Fundamentals. Logo is designed to be conversational. The prompt of its read-eval loop is a question mark (`?`), evoking the question, "what shall I do next?" A natural starting point is to ask Logo to `print` a number:

```
? print 5
5
```

The Logo language employs an unusual call expression syntax that has no delimiting punctuation at all. Above, the argument `5` is passed to `print`, which prints out its argument. The terminology used to describe the programming constructs of Logo differs somewhat from that of Python. Logo has *procedures* rather than the equivalent "functions" in Python, and procedures *output* values rather than "returning" them. The `print` procedure

always outputs `None`, but prints a string representation of its argument as a side effect. (Procedure arguments are typically called *inputs* in Logo, but we will continue to call them arguments in this text for the sake of clarity.)

The most common data type in Logo is a *word*, a string without spaces. Words serve as general-purpose values that can represent numbers, names, and boolean values. Tokens that can be interpreted as numbers or boolean values, such as `5`, evaluate to words directly. On the other hand, names such as `five` are interpreted as procedure calls:

```
? 5
You do not say what to do with 5.
? five
I do not know how to five.
```

While `5` and `five` are interpreted differently, the Logo read-eval loop complains either way. The issue with the first case is that Logo complains whenever a top-level expression it evaluates does not evaluate to `None`. Here, we see the first structural difference between the interpreters for Logo and Calculator; the interface to the former is a read-eval loop that expects the user to print results. The latter employed a more typical read-eval-print loop that printed return values automatically. Python takes a hybrid approach: only non-`None` values are coerced to strings using `repr` and then printed automatically.

A line of Logo can contain multiple expressions in sequence. The interpreter will evaluate each one in turn. It will complain if any top-level expression in a line does not evaluate to `None`. Once an error occurs, the rest of the line is ignored:

```
? print 1 print 2
1
2
? 3 print 4
You do not say what to do with 3.
```

Logo call expressions can be nested. In the version of Logo we will implement, each procedure takes a fixed number of arguments. Therefore, the Logo interpreter is able to determine uniquely when the operands of a nested call expression are complete. Consider, for instance, two procedures `sum` and `difference` that output the sum and difference of their two arguments, respectively:

```
? print sum 10 difference 7 3
14
```

We can see from this nesting example that the parentheses and commas that delimit call expressions are not strictly necessary. In the Calculator interpreter, punctuation allowed us to build expression trees as a purely syntactic operation; without ever consulting the

meaning of the operator names. In Logo, we must use our knowledge of how many arguments each procedure takes in order to discover the correct structure of a nested expression. This issue is addressed in further detail in the next section.

Logo also supports infix operators, such as `+` and `*`. The precedence of these operators is resolved according to the standard rules of algebra; multiplication and division take precedence over addition and subtraction:

```
? 2 + 3 * 4
14
```

The details of how to implement operator precedence and infix operators to form correct expression trees is left as an exercise. For the following discussion, we will concentrate on call expressions using prefix syntax.

Quotation. A bare name is interpreted as the beginning of a call expression, but we would also like to reference words as data. A token that begins with a double quote is interpreted as a word literal. Note that word literals do not have a trailing quotation mark in Logo:

```
? print "hello
hello
```

In dialects of Lisp (and Logo is such a dialect), any expression that is not evaluated is said to be *quoted*. This notion of quotation is derived from a classic philosophical distinction between a thing, such as a dog, which runs around and barks, and the word "dog" that is a linguistic construct for designating such things. When we use "dog" in quotation marks, we do not refer to some dog in particular but instead to a word. In language, quotation allow us to talk about language itself, and so it is in Logo. We can refer to the procedure for `sum` by name without actually applying it by quoting it:

```
? print "sum
sum
```

In addition to words, Logo includes the *sentence* type, interchangeably called a list. Sentences are enclosed in square brackets. The `print` procedure does not show brackets to preserve the conversational style of Logo, but the square brackets can be printed in the output by using the `show` procedure:

```
? print [hello world]
hello world
? show [hello world]
[hello world]
```

Sentences can be constructed using three different two-argument procedures. The `sentence` procedure combines its arguments into a sentence. It is polymorphic; it places its arguments into a new sentence if they are words or concatenates its arguments if they are sentences. The result is always a sentence:

```
? show sentence 1 2
[1 2]
? show sentence 1 [2 3]
[1 2 3]
? show sentence [1 2] 3
[1 2 3]
? show sentence [1 2] [3 4]
[1 2 3 4]
```

The `list` procedure creates a sentence from two elements, which allows the user to create hierarchical data structures:

```
? show list 1 2
[1 2]
? show list 1 [2 3]
[1 [2 3]]
? show list [1 2] 3
[[1 2] 3]
? show list [1 2] [3 4]
[[1 2] [3 4]]
```

Finally, the `fput` procedure creates a list from a first element and the rest of the list, as did the `Rlist` Python constructor from earlier in the chapter:

```
? show fput 1 [2 3]
[1 2 3]
? show fput [1 2] [3 4]
[[1 2] 3 4]
```

Collectively, we can call `sentence` , `list` , and `fput` the *sentence constructors* in Logo. Deconstructing a sentence into its `first` , `last` , and rest (called `butfirst`) in Logo is straightforward as well. Hence, we also have a set of selector procedures for sentences:

```
? print first [1 2 3]
1
? print last [1 2 3]
3
? print butfirst [1 2 3]
[2 3]
```

Expressions as Data. The contents of a sentence is also quoted in the sense that it is not evaluated. Hence, we can print Logo expressions without evaluating them:

```
? show [print sum 1 2]
[print sum 1 2]
```

The purpose of representing Logo expressions as sentences is typically not to print them out, but instead to evaluate them using the `run` procedure:

```
? run [print sum 1 2]
3
```

Combining quotation, sentence constructors, and the `run` procedure, we arrive at a very general means of combination that builds Logo expressions on the fly and then evaluates them:

```
? run sentence "print [sum 1 2]
3
? print run sentence "sum sentence 10 run [difference 7 3]
14
```

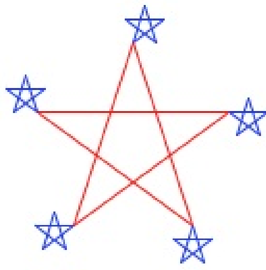
The point of this last example is to show that while the procedures `sum` and `difference` are not first-class constructs in Logo (they cannot be placed in a sentence, for instance), their quoted names are first-class, and the `run` procedure can resolve those names to the procedures to which they refer.

The ability to represent code as data and later interpret it as part of the program is a defining feature of Lisp-style languages. The idea that a program can rewrite itself as it executes is a powerful one, and served as the foundation for early research in artificial intelligence (AI). Lisp was the preferred language of AI researchers for decades. The Lisp language was invented by John McCarthy, who coined the term "artificial intelligence" and played a critical role in defining the field. This code-as-data property of Lisp dialects, along with their simplicity and elegance, continues to attract new Lisp programmers today.

Turtle graphics. No implementation of Logo is complete without graphical output based on the Logo turtle. This turtle begins in the center of a canvas, moves and turns based on procedures, and draws lines behind it as it moves. While the turtle was invented to engage children in the act of programming, it remains an entertaining graphical tool for even advanced programmers.

At any moment during the course of executing a Logo program, the Logo turtle has a position and heading on the canvas. Single-argument procedures such as `forward` and `right` change the position and heading of the turtle. Common procedures have abbreviations: `forward` can also be called as `fd`, etc. The nested expression below draws a star with a smaller star at each vertex:

```
? repeat 5 [fd 100 repeat 5 [fd 20 rt 144] rt 144]
```



The full repertoire of Turtle procedures is also built into Python as the turtle library module. A limited subset of these functions are exposed as Logo procedures in the companion project to this chapter.

Assignment. Logo supports binding names to values. As in Python, a Logo environment consists of a sequence of frames, and each frame can have at most one value bound to a given name. In Logo, names are bound with the `make` procedure, which takes as arguments a name and a value:

```
? make "x 2
```

The first argument is the name `x`, rather than the output of applying the procedure `x`, and so it must be quoted. The values bound to names are retrieved by evaluating expressions that begin with a colon:

```
? print :x
2
```

Any word that begins with a colon, such as `:x`, is called a variable. A variable evaluates to the value to which the name of the variable is bound in the current environment.

The `make` procedure does not have the same effect as an assignment statement in Python. The name passed to `make` is either already bound to a value or is currently unbound.

1. If the name is already bound, `make` re-binds that name in the first frame in which it is found.
2. If the name is not bound, `make` binds the name in the global frame.

This behavior contrasts sharply with the semantics of the Python assignment statement, which always binds a name to a value in the first frame of the current environment. The first assignment rule above is similar to Python assignment following a `nonlocal` statement. The second is similar to Python assignment following a `global` statement.

Procedures. Logo supports user-defined procedures using definitions that begin with the `to` keyword. Definitions are the final type of expression in Logo, along with call expressions, primitive expressions, and quoted expressions. The first line of a definition gives the name of the new procedure, followed by the formal parameters as variables. The

lines that follow constitute the body of the procedure, which can span multiple lines and must end with a line that contains only the token `end`. The Logo read-eval loop prompts the user for procedure bodies with a `>` continuation symbol. Values are output from a user-defined procedure using the `output` procedure:

```
? to double :x
> output sum :x :x
> end
? print double 4
8
```

Logo's application process for a user-defined procedure is similar to the process in Python. Applying a procedure to a sequence of arguments begins by extending an environment with a new frame, binding the formal parameters of the procedure to the argument values, and then evaluating the lines of the body of the procedure in the environment that starts with that new frame.

A call to `output` has the same role in Logo as a `return` statement in Python: it halts the execution of the body of a procedure and returns a value. A Logo procedure can return no value at all by calling `stop`:

```
? to count
> print 1
> print 2
> stop
> print 3
> end
? count
1
2
```

Scope. Logo is a *dynamically scoped* language. A lexically scoped language such as Python does not allow the local names of one function to affect the evaluation of another function unless the second function was explicitly defined within the first. The formal parameters of two top-level functions are completely isolated. In a dynamically scoped language, there is no such isolation. When one function calls another function, the names bound in the local frame for the first are accessible in the body of the second:

```
? to print_last_x
> print :x
> end
? to print_x :x
> print_last_x
> end
? print_x 5
5
```

While the name `x` is not bound in the global frame, it is bound in the local frame for `print_x`, the function that is called first. Logo's dynamic scoping rules allow the function `print_last_x` to refer to `x`, which was bound as the formal parameter of `print_x`.

Dynamic scoping is implemented by a single change to the environment model of computation. The frame that is created by calling a user-defined function always extends the current environment. For example, the call to `print_x` above introduces a new frame that extends the current environment, which consists solely of the global frame. Within the body of `print_x`, the call to `print_last_x` introduces another frame that extends the current environment, which includes both the local frame for `print_x` and the global frame. As a result, looking up the name `x` in the body of `print_last_x` finds that name bound to `5` in the local frame for `print_x`. Alternatively, under the lexical scoping rules of Python, the frame for `print_last_x` would have extended only the global frame and not the local frame for `print_x`.

A dynamically scoped language has the advantage that its procedures may not need to take as many arguments. For instance, the `print_last_x` procedure above takes no arguments, and yet its behavior can be parameterized by an enclosing scope.

General programming. Our tour of Logo is complete, and yet we have not introduced any advanced features, such as an object system, higher-order procedures, or even statements. Learning to program effectively in Logo requires piecing together the simple features of the language into effective combinations.

There is no conditional expression type in Logo; the procedures `if` and `ifelse` are applied using call expression evaluation rules. The first argument of `if` is a boolean word, either `True` or `False`. The second argument is not an output value, but instead a sentence that contains the line of Logo code to be evaluated if the first argument is `True`. An important consequence of this design is that the contents of the second argument is not evaluated at all unless it will be used:

```
? 1/0
div raised a ZeroDivisionError: division by zero
? to reciprocal :x
> if not :x = 0 [output 1 / :x]
> output "infinity
> end
? print reciprocal 2
0.5
? print reciprocal 0
infinity
```

Not only does the Logo conditional expression not require a special syntax, but it can in fact be implemented in terms of `word` and `run`. The primitive procedure `ifelse` takes three arguments: a boolean word, a sentence to be evaluated if that word is `True`, and a

sentence to be evaluated if that word is `False`. By clever naming of the formal parameters, we can implement a user-defined procedure `ifelse2` with the same behavior:

```
? to ifelse2 :predicate :True :False
> output run run word ": :predicate
> end
? print ifelse2 emptyp [] ["empty] ["full]
empty
```

Recursive procedures do not require any special syntax, and they can be used with `run`, `sentence`, `first`, and `butfirst` to define general sequence operations on sentences. For instance, we can apply a procedure to an argument by building a two-element sentence and running it. The argument must be quoted if it is a word:

```
? to apply_fn :fn :arg
> output run list :fn ifelse word? :arg [word "" :arg] [:arg]
> end
```

Next, we can define a procedure for mapping a procedure `:fn` over the words in a sentence `:s` incrementally:

```
? to map_fn :fn :s
> if emptyp :s [output []]
> output fput apply_fn :fn first :s map_fn :fn butfirst :s
> end
? show map "double [1 2 3]
[2 4 6]
```

The second line of the body of `map_fn` can also be written with parentheses to indicate the nested structure of the call expression. However, parentheses show where call expressions begin and end, rather than surrounding only the operands and not the operator:

```
> (output (fput (apply_fn :fn (first :s)) (map_fn :fn (butfirst :s))))
```

Parentheses are not necessary in Logo, but they often assist programmers in documenting the structure of nested expressions. Most dialects of Lisp require parentheses and therefore have a syntax with explicit nesting.

As a final example, Logo can express recursive drawings using its turtle graphics in a remarkably compact form. Sierpinski's triangle is a fractal that draws each triangle as three neighboring triangles that have vertexes at the midpoints of the legs of the triangle that contains them. It can be drawn to a finite recursive depth by this Logo program:

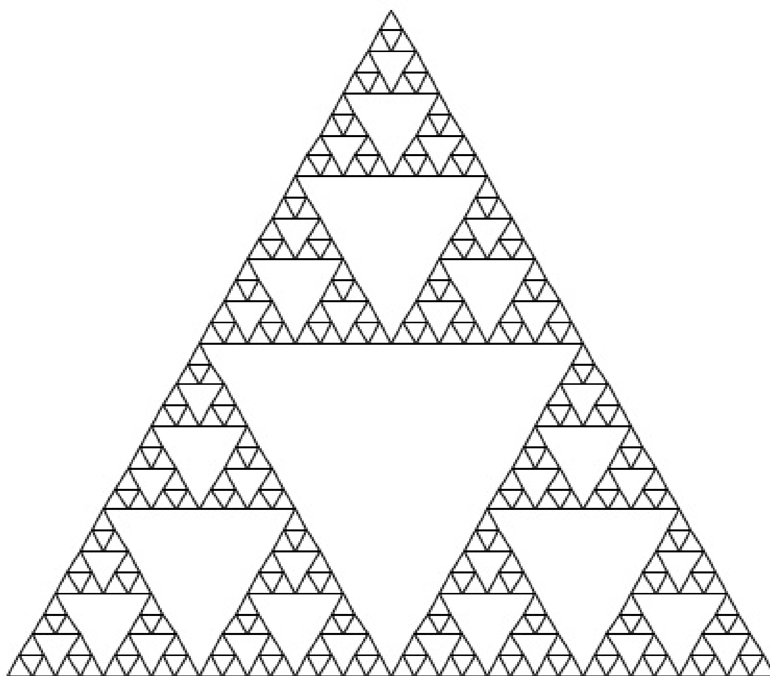
```
? to triangle :exp
> repeat 3 [run :exp lt 120]
> end

? to sierpinski :d :k
> triangle [ifelse :k = 1 [fd :d] [leg :d :k]]
> end

? to leg :d :k
> sierpinski :d / 2 :k - 1
> penup fd :d pendown
> end
```

The `triangle` procedure is a general method for repeating a drawing procedure three times with a left turn following each repetition. The `sierpinski` procedure takes a length and a recursive depth. It draws a plain triangle if the depth is `1`, and otherwise draws a triangle made up of calls to `leg`. The `leg` procedure draws a single leg of a recursive Sierpinski triangle by a recursive call to `sierpinski` that fills the first half of the length of the leg, then by moving the turtle to the next vertex. The procedures `up` and `down` stop the turtle from drawing as it moves by lifting its pen up and the placing it down again. The mutual recursion between `sierpinski` and `leg` yields this result:

```
? sierpinski 400 6
```



3.6.3 Structure

This section describes the general structure of a Logo interpreter. While this chapter is self-contained, it does reference the companion project. Completing that project will produce a working implementation of the interpreter sketch described here.

An interpreter for Logo can share much of the same structure as the Calculator interpreter. A parser produces an expression data structure that is interpreted by an evaluator. The evaluation function inspects the form of an expression, and for call expressions it calls a function to apply a procedure to some arguments. However, there are structural differences that accommodate Logo's unusual syntax.

Lines. The Logo parser does not read a single expression, but instead reads a full line of code that may contain multiple expressions in sequence. Rather than returning an expression tree, it returns a Logo sentence.

The parser actually does very little syntactic analysis. In particular, parsing does not differentiate the operator and operand subexpressions of call expressions into different branches of a tree. Instead, the components of a call expression are listed in sequence, and nested call expressions are represented as a flat sequence of tokens. Finally, parsing does not determine the type of even primitive expressions such as numbers because Logo does not have a rich type system; instead, every element is a word or a sentence.

```
>>> parse_line('print sum 10 difference 7 3')
['print', 'sum', '10', 'difference', '7', '3']
```

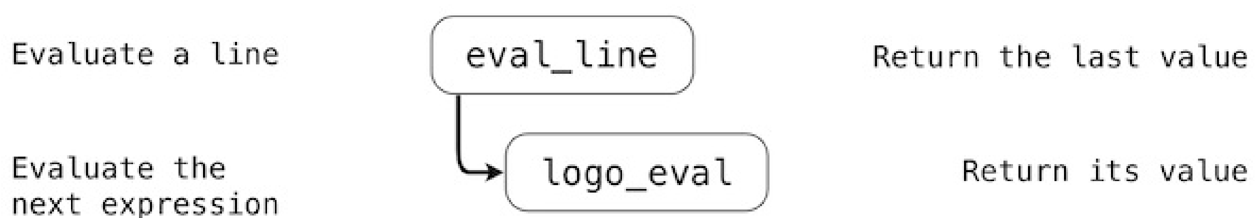
The parser performs so little analysis because the dynamic character of Logo requires that the evaluator resolve the structure of nested expressions.

The parser does identify the nested structure of sentences. Sentences within sentences are represented as nested Python lists.

```
>>> parse_line('print sentence "this [is a [deep] list]')
['print', 'sentence', '"this', ['is', 'a', ['deep'], 'list']]
```

A complete implementation of `parse_line` appears in the companion projects as `logo_parser.py`.

Evaluation. Logo is evaluated one line at a time. A skeleton implementation of the evaluator is defined in `logo.py` of the companion project. The sentence returned from `parse_line` is passed to the `eval_line` function, which evaluates each expression in the line. The `eval_line` function repeatedly calls `logo_eval`, which evaluates the next full expression in the line until the line has been evaluated completely, then returns the last value. The `logo_eval` function evaluates a single expression.



The `logo_eval` function evaluates the different forms of expressions that we introduced in the last section: primitives, variables, definitions, quoted expressions, and call expressions. The form of a multi-element expression in Logo can be determined by inspecting its first element. Each form of expression as its own evaluation rule.

1. A primitive expression (a word that can be interpreted as a number, `True`, or `False`) evaluates to itself.
2. A variable is looked up in the environment. Environments are discussed in detail in the next section.
3. A definition is handled as a special case. User-defined procedures are also discussed in the next section.
4. A quoted expression evaluates to the text of the quotation, which is a string without the preceding quote. Sentences (represented as Python lists) are also considered to be quoted; they evaluate to themselves.
5. A call expression looks up the operator name in the current environment and applies the procedure that is bound to that name.

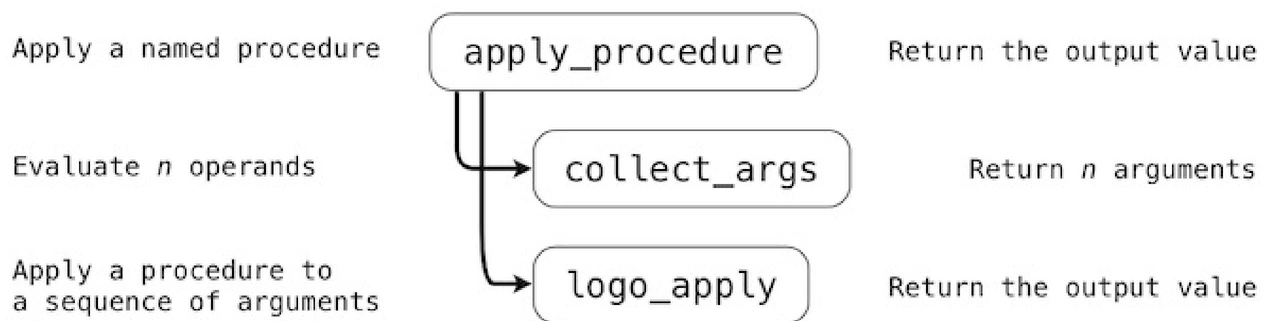
A simplified implementation of `logo_apply` appears below. Some error checking has been removed in order to focus our discussion. A more robust implementation appears in the companion project.

```
>>> def logo_eval(line, env):
        """Evaluate the first expression in a line."""
        token = line.pop()
        if isprimitive(token):
            return token
        elif isvariable(token):
            return env.lookup_variable(variable_name(token))
        elif isdefinition(token):
            return eval_definition(line, env)
        elif isquoted(token):
            return text_of_quotation(token)
        else:
            procedure = env.procedures.get(token, None)
            return apply_procedure(procedure, line, env)
```

The final case above invokes a second process, procedure application, that is expressed by a function `apply_procedure`. To apply a procedure named by an operator token, that operator is looked up in the current environment. In the definition above, `env` is an instance of the `Environment` class described in the next section. The attribute `env.procedures` is a dictionary that stores the mapping between operator names and procedures. In Logo, an environment has a single such mapping; there are no locally defined procedures. Moreover, Logo maintains separate mappings, called separate *namespaces*, for the the names of procedures and the names of variables. A procedure and an unrelated variable can have the same name in Logo. However, reusing names in this way is not recommended.

Procedure application. Procedure application begins by calling the `apply_procedure` function, which is passed the procedure looked up by `logo_apply`, along with the remainder of the current line of code and the current environment. The procedure application process in Logo is considerably more general than the `calc_apply` function in Calculator. In particular, `apply_procedure` must inspect the procedure it is meant to apply in order to determine its argument count *n*, before evaluating *n* operand expressions. It is here that we see why the Logo parser was unable to build an expression tree by syntactic analysis alone; the structure of the tree is determined by the procedure.

The `apply_procedure` function calls a function `collect_args` that must repeatedly call `logo_eval` to evaluate the next *n* expressions on the line. Then, having computed the arguments to the procedure, `apply_procedure` calls `logo_apply`, the function that actually applies procedures to arguments. The call graph below illustrates the process.



The final function `logo_apply` applies two kinds of arguments: primitive procedures and user-defined procedures, both of which are instances of the `Procedure` class. A `Procedure` is a Python object that has instance attributes for the name, argument count, body, and formal parameters of a procedure. The type of the `body` attribute varies. A primitive procedure is implemented in Python, and so its `body` is a Python function. A user-defined (non-primitive) procedure is defined in Logo, and so its `body` is a list of lines of Logo code. A `Procedure` also has two boolean-valued attributes, one to indicated whether it is primitive and another to indicate whether it needs access to the current environment.

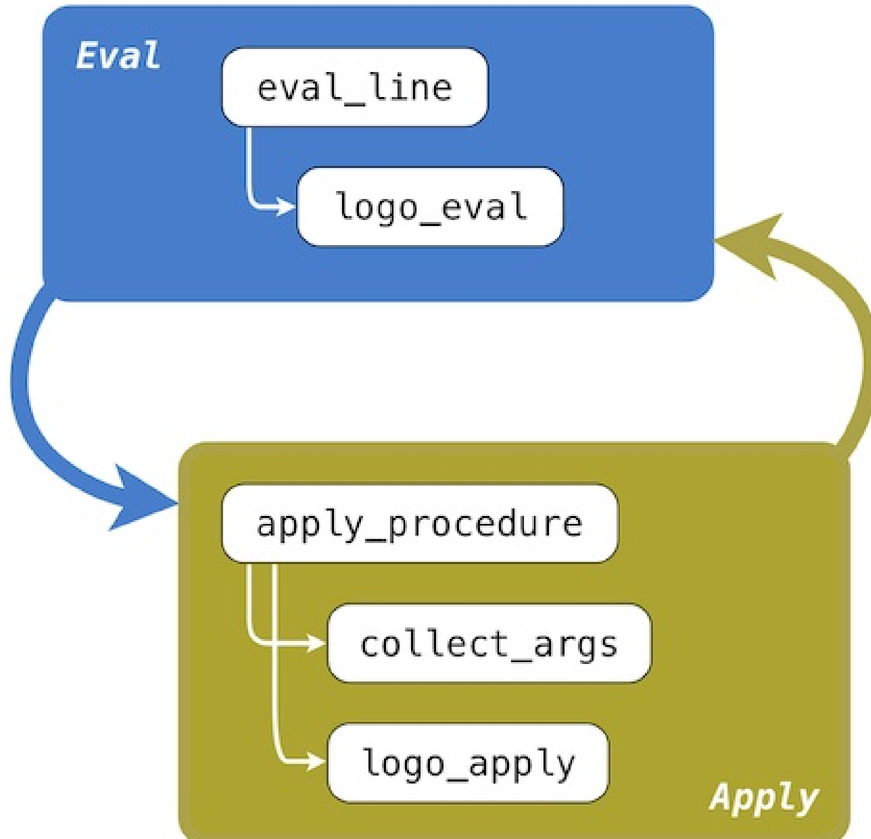
```
>>> class Procedure():
        def __init__(self, name, arg_count, body, isprimitive=False,
                     needs_env=False, formal_params=None):
            self.name = name
            self.arg_count = arg_count
            self.body = body
            self.isprimitive = isprimitive
            self.needs_env = needs_env
            self.formal_params = formal_params
```

A primitive procedure is applied by calling its body on the argument list and returning its return value as the output of the procedure.

```
>>> def logo_apply(proc, args):
        """Apply a Logo procedure to a list of arguments."""
        if proc.isprimitive:
            return proc.body(*args)
        else:
            """Apply a user-defined procedure"""
```

The body of a user-defined procedure is a list of lines, each of which is a Logo sentence. To apply the procedure to a list of arguments, we evaluate the lines of the body in a new environment. To construct this environment, a new frame is added to the environment in which the formal parameters of the procedure are bound to the arguments. The important structural aspect of this process is that evaluating a line of the body of a user-defined procedure requires a recursive call to `eval_line` .

Eval/apply recursion. The functions that implement the evaluation process, `eval_line` and `logo_eval` , and the functions that implement the function application process, `apply_procedure` , `collect_args` , and `logo_apply` , are mutually recursive. Evaluation requires application whenever a call expression is found. Application uses evaluation to evaluate operand expressions into arguments, as well as to evaluate the body of user-defined procedures. The general structure of this mutually recursive process appears in interpreters quite generally: evaluation is defined in terms of application and application is defined in terms of evaluation.



This recursive cycle ends with language primitives. Evaluation has a base case that is evaluating a primitive expression, variable, quoted expression, or definition. Function application has a base case that is applying a primitive procedure. This mutually recursive structure, between an eval function that processes expression forms and an apply function that processes functions and their arguments, constitutes the essence of the evaluation process.

3.6.4 Environments

Now that we have described the structure of our Logo interpreter, we turn to implementing the `Environment` class so that it correctly supports assignment, procedure definition, and variable lookup with dynamic scope. An `Environment` instance represents the collective set of name bindings that are accessible at some point in the course of program execution. Bindings are organized into frames, and frames are implemented as Python dictionaries. Frames contain name bindings for variables, but not procedures; the bindings between operator names and `Procedure` instances are stored separately in Logo. In the project implementation, frames that contain variable name bindings are stored as a list of dictionaries in the `_frames` attribute of an `Environment`, while procedure name bindings are stored in the dictionary-valued `procedures` attribute.

Frames are not accessed directly, but instead through two `Environment` methods: `lookup_variable` and `set_variable_value`. The first implements a process identical to the look-up procedure that we introduced in the environment model of computation in Chapter 1. A name is matched against the bindings of the first (most recently added) frame of the current environment. If it is found, the value to which it is bound is returned. If it is not found, look-up proceeds to the frame that was extended by the current frame.

The `set_variable_value` method also searches for a binding that matches a variable name. If one is found, it is updated with a new value. If none is found, then a new binding is created in the global frame. The implementations of these methods are left as an exercise in the companion project.

The `lookup_variable` method is invoked from `logo_eval` when evaluating a variable name. The `set_variable_value` method is invoked by the `logo_make` function, which serves as the body of the primitive `make` procedure in Logo.

```
>>> def logo_make(symbol, val, env):
        """Apply the Logo make primitive, which binds a name to a value."""
        env.set_variable_value(symbol, val)
```

With the addition of variables and the `make` primitive, our interpreter supports its first means of abstraction: binding names to values. In Logo, we can now replicate our first abstraction steps in Python from Chapter 1:

```
? make "radius 10
? print 2 * :radius
20
```

Assignment is only a limited form of abstraction. We have seen from the beginning of this course that user-defined functions are a critical tool in managing the complexity of even moderately sized programs. Two enhancements will enable user-defined procedures in Logo. First, we must describe the implementation of `eval_definition`, the Python function called from `logo_eval` when the current line is a definition. Second, we will complete our description of the process in `logo_apply` that applies a user-defined procedure to some arguments. Both of these changes leverage the `Procedure` class defined in the previous section.

A definition is evaluated by creating a new `Procedure` instance that represents the user-defined procedure. Consider the following Logo procedure definition:

```
? to factorial :n
> output ifelse :n = 1 [1] [:n * factorial :n - 1]
> end
? print fact 5
120
```

The first line of the definition supplies the name `factorial` and formal parameter `n` of the procedure. The line that follows constitute the body of the procedure. This line is not evaluated immediately, but instead stored for future application. That is, the line is read and parsed by `eval_definition`, but not passed to `eval_line`. Lines of the body are read from the user until a line containing only `end` is encountered. In Logo, `end` is not a procedure to be evaluated, nor is it part of the procedure body; it is a syntactic marker of the end of a procedure definition.

The `Procedure` instance created from this procedure name, formal parameter list, and body, is registered in the `procedures` dictionary attribute of the environment. In Logo, unlike Python, once a procedure is bound to a name, no other definition can use that name again.

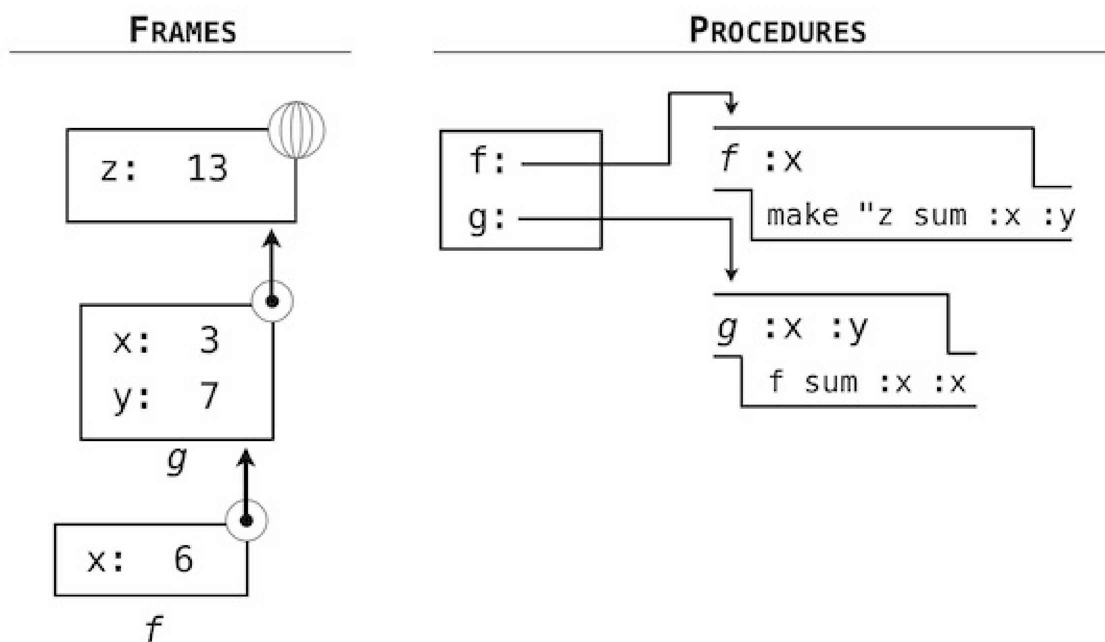
The `logo_apply` function applies a `Procedure` instance to some arguments, which are Logo values represented as strings (for words) and lists (for sentences). For a user-defined procedure, `logo_apply` creates a new frame, a dictionary object in which the the keys are the formal parameters of the procedure and the values are the arguments. In a dynamically scoped language such as Logo, this new frame always extends the current environment in which the procedure was called. Therefore, we append the newly created frame onto the

current environment. Then, each line of the body is passed to `eval_line` in turn. Finally, we can remove the newly created frame from the environment after evaluating its body. Because Logo does not support higher-order or first-class procedures, we never need to track more than one environment at a time throughout the course of execution of a program.

The following example illustrates the list of frames and dynamic scoping rules that result from applying these two user-defined Logo procedures:

```
? to f :x
> make "z sum :x :y
> end
? to g :x :y
> f sum :x :x
> end
? g 3 7
? print :z
13
```

The environment created from the evaluation of these expressions is divided between procedures and frames, which are maintained in separate name spaces. The order of frames is determined by the order of calls.



3.6.5 Data as Programs

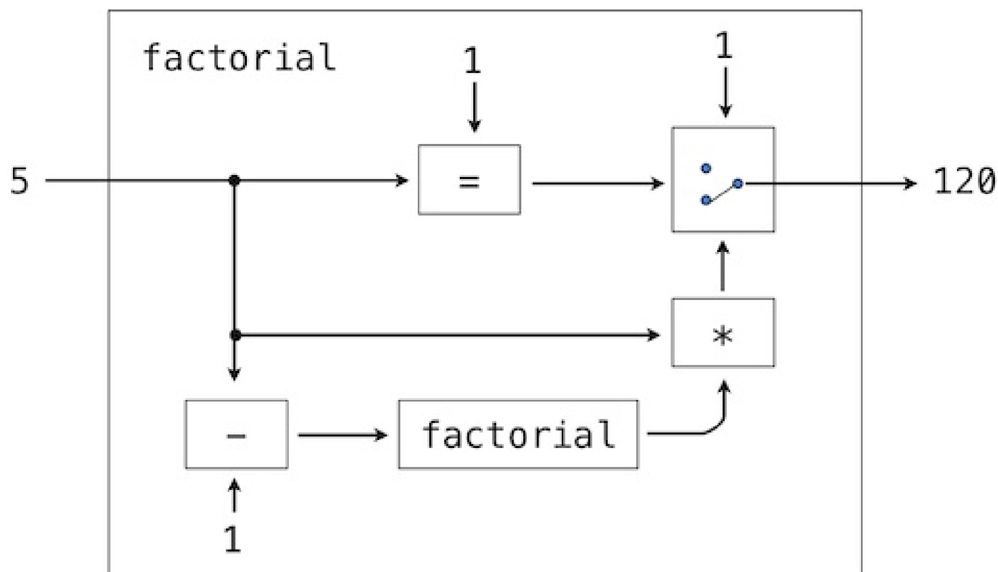
In thinking about a program that evaluates Logo expressions, an analogy might be helpful. One operational view of the meaning of a program is that a program is a description of an abstract machine. For example, consider again this procedure to compute factorials:

```
? to factorial :n
> output ifelse :n = 1 [1] [:n * factorial :n - 1]
> end
```

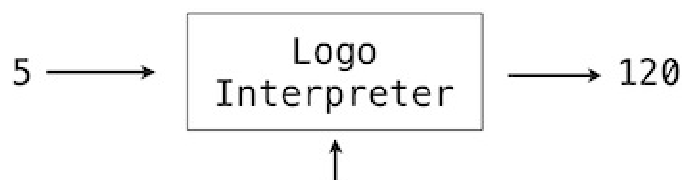
We could express an equivalent program in Python as well, using a conditional expression.

```
>>> def factorial(n):
        return 1 if n == 1 else n * factorial(n - 1)
```

We may regard this program as the description of a machine containing parts that decrement, multiply, and test for equality, together with a two-position switch and another factorial machine. (The factorial machine is infinite because it contains another factorial machine within it.) The figure below is a flow diagram for the factorial machine, showing how the parts are wired together.



In a similar way, we can regard the Logo interpreter as a very special machine that takes as input a description of a machine. Given this input, the interpreter configures itself to emulate the machine described. For example, if we feed our evaluator the definition of factorial the evaluator will be able to compute factorials.




```
to factorial :n
output ifelse :n = 1 [1] [:n * factorial :n - 1]
end
```

From this perspective, our Logo interpreter is seen to be a universal machine. It mimics other machines when these are described as Logo programs. It acts as a bridge between the data objects that are manipulated by our programming language and the programming language itself. Image that a user types a Logo expression into our running Logo interpreter. From the perspective of the user, an input expression such as `sum 2 2` is an expression in

the programming language, which the interpreter should evaluate. From the perspective of the Logo interpreter, however, the expression is simply a sentence of words that is to be manipulated according to a well-defined set of rules.

That the user's programs are the interpreter's data need not be a source of confusion. In fact, it is sometimes convenient to ignore this distinction, and to give the user the ability to explicitly evaluate a data object as an expression. In Logo, we use this facility whenever employing the `run` procedure. Similar functions exist in Python: the `eval` function will evaluate a Python expression and the `exec` function will execute a Python statement. Thus,

```
>>> eval('2+2')
4
```

and

```
>>> 2+2
4
```

both return the same result. Evaluating expressions that are constructed as a part of execution is a common and powerful feature in dynamic programming languages. In few languages is this practice as common as in Logo, but the ability to construct and evaluate expressions during the course of execution of a program can prove to be a valuable tool for any programmer.

Chapter 4: Distributed and Parallel Computing

Contents

4.1 Introduction

So far, we have focused on how to create, interpret, and execute programs. In Chapter 1, we learned to use functions as a means for combination and abstraction. Chapter 2 showed us how to represent data and manipulate it with data structures and objects, and introduced us to the concept of data abstraction. In Chapter 3, we learned how computer programs are interpreted and executed. The result is that we understand how to design programs for a single processor to run.

In this chapter, we turn to the problem of coordinating multiple computers and processors. First, we will look at distributed systems. These are interconnected groups of independent computers that need to communicate with each other to get a job done. They may need to coordinate to provide a service, share data, or even store data sets that are too large to fit on a single machine. We will look at different roles computers can play in distributed systems and learn about the kinds of information that computers need to exchange in order to work together.

Next, we will consider concurrent computation, also known as parallel computation. Concurrent computation is when a single program is executed by multiple processors with a shared memory, all working together in parallel in order to get work done faster. Concurrency introduces new challenges, and so we will develop new techniques to manage the complexity of concurrent programs.

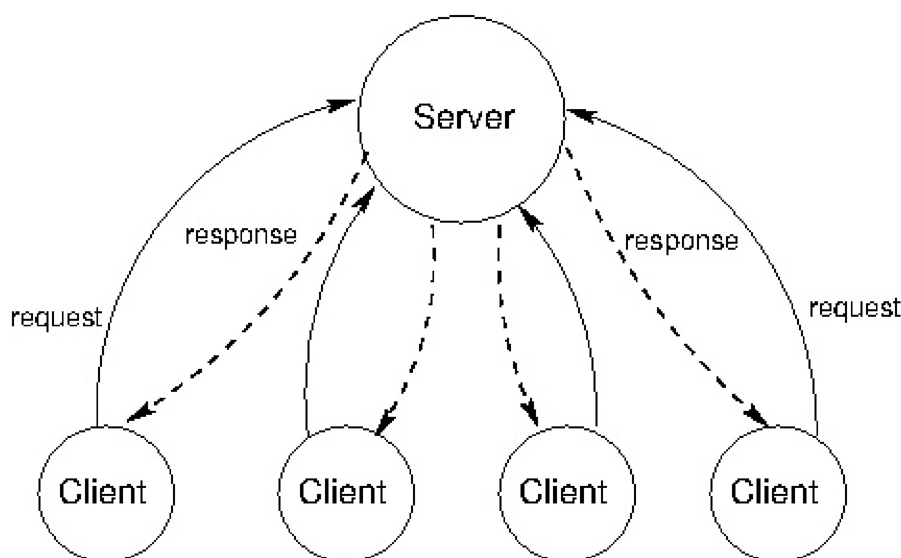
4.2 Distributed Computing

A distributed system is a network of autonomous computers that communicate with each other in order to achieve a goal. The computers in a distributed system are independent and do not physically share memory or processors. They communicate with each other using *messages*, pieces of information transferred from one computer to another over a network. Messages can communicate many things: computers can tell other computers to execute a procedures with particular arguments, they can send and receive packets of data, or send signals that tell other computers to behave a certain way.

Computers in a distributed system can have different roles. A computer's role depends on the goal of the system and the computer's own hardware and software properties. There are two predominant ways of organizing computers in a distributed system. The first is the client-server architecture, and the second is the peer-to-peer architecture.

4.2.1 Client/Server Systems

The client-server architecture is a way to dispense a service from a central source. There is a single *server* that provides a service, and multiple *clients* that communicate with the server to consume its products. In this architecture, clients and servers have different jobs. The server's job is to respond to service requests from clients, while a client's job is to use the data provided in response in order to perform some task.



The client-server model of communication can be traced back to the introduction of UNIX in the 1970's, but perhaps the most influential use of the model is the modern World Wide Web. An example of a client-server interaction is reading the New York Times online. When the web server at www.nytimes.com is contacted by a web browsing client (like Firefox), its

job is to send back the HTML of the New York Times main page. This could involve calculating personalized content based on user account information sent by the client, and fetching appropriate advertisements. The job of the web browsing client is to render the HTML code sent by the server. This means displaying the images, arranging the content visually, showing different colors, fonts, and shapes and allowing users to interact with the rendered web page.

The concepts of *client* and *server* are powerful functional abstractions. A server is simply a unit that provides a service, possibly to multiple clients simultaneously, and a client is a unit that consumes the service. The clients do not need to know the details of how the service is provided, or how the data they are receiving is stored or calculated, and the server does not need to know how the data is going to be used.

On the web, we think of clients and servers as being on different machines, but even systems on a single machine can have client/server architectures. For example, signals from input devices on a computer need to be generally available to programs running on the computer. The programs are clients, consuming mouse and keyboard input data. The operating system's device drivers are the servers, taking in physical signals and serving them up as usable input.

A drawback of client-server systems is that the server is a single point of failure. It is the only component with the ability to dispense the service. There can be any number of clients, which are interchangeable and can come and go as necessary. If the server goes down, however, the system stops working. Thus, the functional abstraction created by the client-server architecture also makes it vulnerable to failure.

Another drawback of client-server systems is that resources become scarce if there are too many clients. Clients increase the demand on the system without contributing any computing resources. Client-server systems cannot shrink and grow with changing demand.

4.2.2 Peer-to-peer Systems

The client-server model is appropriate for service-oriented situations. However, there are other computational goals for which a more equal division of labor is a better choice. The term *peer-to-peer* is used to describe distributed systems in which labor is divided among all the components of the system. All the computers send and receive data, and they all contribute some processing power and memory. As a distributed system increases in size, its capacity of computational resources increases. In a peer-to-peer system, all components of the system contribute some processing power and memory to a distributed computation.

Division of labor among *all* participants is the identifying characteristic of a peer-to-peer system. This means that peers need to be able to communicate with each other reliably. In order to make sure that messages reach their intended destinations, peer-to-peer systems need to have an organized network structure. The components in these systems cooperate to maintain enough information about the locations of other components to send messages to intended destinations.

In some peer-to-peer systems, the job of maintaining the health of the network is taken on by a set of specialized components. Such systems are not pure peer-to-peer systems, because they have different types of components that serve different functions. The components that support a peer-to-peer network act like scaffolding: they help the network stay connected, they maintain information about the locations of different computers, and they help newcomers take their place within their neighborhood.

The most common applications of peer-to-peer systems are data transfer and data storage. For data transfer, each computer in the system contributes to send data over the network. If the destination computer is in a particular computer's neighborhood, that computer helps send data along. For data storage, the data set may be too large to fit on any single computer, or too valuable to store on just a single computer. Each computer stores a small portion of the data, and there may be multiple copies of the same data spread over different computers. When a computer fails, the data that was on it can be restored from other copies and put back when a replacement arrives.

Skype, the voice- and video-chat service, is an example of a data transfer application with a peer-to-peer architecture. When two people on different computers are having a Skype conversation, their communications are broken up into packets of 1s and 0s and transmitted through a peer-to-peer network. This network is composed of other people whose computers are signed into Skype. Each computer knows the location of a few other computers in its neighborhood. A computer helps send a packet to its destination by passing it on a neighbor, which passes it on to some other neighbor, and so on, until the packet reaches its intended destination. Skype is not a pure peer-to-peer system. A scaffolding network of *supernodes* is responsible for logging-in and logging-out users, maintaining information about the locations of their computers, and modifying the network structure to deal with users entering and leaving.

4.2.3 Modularity

The two architectures we have just considered -- peer-to-peer and client-server -- are designed to enforce *modularity*. Modularity is the idea that the components of a system should be black boxes with respect to each other. It should not matter how a component

implements its behavior, as long as it upholds an *interface*: a specification for what outputs will result from inputs.

In chapter 2, we encountered interfaces in the context of dispatch functions and object-oriented programming. There, interfaces took the form of specifying the messages that objects should take, and how they should behave in response to them. For example, in order to uphold the "representable as strings" interface, an object must be able to respond to the `__repr__` and `__str__` messages, and output appropriate strings in response. How the generation of those strings is implemented is not part of the interface.

In distributed systems, we must consider program design that involves multiple computers, and so we extend this notion of an interface from objects and messages to full programs. An interface specifies the inputs that should be accepted and the outputs that should be returned in response to inputs. Interfaces are everywhere in the real world, and we often take them for granted. A familiar example is TV remotes. You can buy many different brands of remote for a modern TV, and they will all work. The only commonality between them is the "TV remote" interface. A piece of electronics obeys the "TV remote" interface as long as it sends the correct signals to your TV (the output) in response to when you press the power, volume, channel, or whatever other buttons (the input).

Modularity gives a system many advantages, and is a property of thoughtful system design. First, a modular system is easy to understand. This makes it easier to change and expand. Second, if something goes wrong with the system, only the defective components need to be replaced. Third, bugs or malfunctions are easy to localize. If the output of a component doesn't match the specifications of its interface, even though the inputs are correct, then that component is the source of the malfunction.

4.2.4 Message Passing

In distributed systems, components communicate with each other using message passing. A message has three essential parts: the sender, the recipient, and the content. The sender needs to be specified so that the recipient knows which component sent the message, and where to send replies. The recipient needs to be specified so that any computers who are helping send the message know where to direct it. The content of the message is the most variable. Depending on the function of the overall system, the content can be a piece of data, a signal, or instructions for the remote computer to evaluate a function with some arguments.

This notion of message passing is closely related to the message passing technique from Chapter 2, in which dispatch functions or dictionaries responded to string-valued messages. Within a program, the sender and receiver are identified by the rules of evaluation. In a distributed system however, the sender and receiver must be explicitly encoded in the

message. Within a program, it is convenient to use strings to control the behavior of the dispatch function. In a distributed system, messages may need to be sent over a network, and may need to hold many different kinds of signals as 'data', so they are not always encoded as strings. In both cases, however, messages serve the same function. Different components (dispatch functions or computers) exchange them in order to achieve a goal that requires coordinating multiple modular components.

At a high level, message contents can be complex data structures, but at a low level, messages are simply streams of 1s and 0s sent over a network. In order to be usable, all messages sent over a network must be formatted according to a consistent *message protocol*.

A **message protocol** is a set of rules for encoding and decoding messages. Many message protocols specify that a message conform to a particular format, in which certain bits have a consistent meaning. A fixed format implies fixed encoding and decoding rules to generate and read that format. All the components in the distributed system must understand the protocol in order to communicate with each other. That way, they know which part of the message corresponds to which information.

Message protocols are not particular programs or software libraries. Instead, they are rules that can be applied by a variety of programs, even written in different programming languages. As a result, computers with vastly different software systems can participate in the same distributed system, simply by conforming to the message protocols that govern the system.

4.2.5 Messages on the World Wide Web

HTTP (short for Hypertext Transfer Protocol) is the message protocol that supports the world wide web. It specifies the format of messages exchanged between a web browser and a web server. All web browsers use the HTTP format to request pages from a web server, and all web servers use the HTTP format to send back their responses.

When you type in a URL into your web browser, say http://en.wikipedia.org/wiki/UC_Berkeley , you are in fact telling your browser that it must request the page "wiki/UC_Berkeley" from the server called "en.wikipedia.org" using the "http" protocol. The sender of the message is your computer, the recipient is en.wikipedia.org, and the format of the message content is:

```
GET /wiki/UC_Berkeley HTTP/1.1
```

The first word is the type of the request, the next word is the resource that is requested, and after that is the name of the protocol (HTTP) and the version (1.1). (There are another types of requests, such as PUT, POST, and HEAD, that web browsers can also use).

The server sends back a reply. This time, the sender is en.wikipedia.org, the recipient is your computer, and the format of the message content is a header, followed by data:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2011 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2011 23:11:55 GMT
Content-Type: text/html; charset=UTF-8

... web page content ...
```

On the first line, the words "200 OK" mean that there were no errors. The subsequent lines of the header give information about the server, the date, and the type of content being sent back. The header is separated from the actual content of the web page by a blank line.

If you have typed in a wrong web address, or clicked on a broken link, you may have seen a message like this error:

```
404 Error File Not Found
```

It means that the server sent back an HTTP header that started like this:

```
HTTP/1.1 404 Not Found
```

A fixed set of response codes is a common feature of a message protocol. Designers of protocols attempt to anticipate common messages that will be sent via the protocol and assign fixed codes to reduce transmission size and establish a common message semantics. In the HTTP protocol, the 200 response code indicates success, while 404 indicates an error that a resource was not found. A variety of other response codes exist in the HTTP 1.1 standard as well.

HTTP is a fixed format for communication, but it allows arbitrary web pages to be transmitted. Other protocols like this on the internet are XMPP, a popular protocol for instant messages, and FTP, a protocol for downloading and uploading files between client and server.

4.3 Parallel Computing

Computers get faster and faster every year. In 1965, Intel co-founder Gordon Moore made a prediction about how much faster computers would get with time. Based on only five data points, he extrapolated that the number of transistors that could inexpensively be fit onto a chip would double every two years. Almost 50 years later, his prediction, now called Moore's law, remains startlingly accurate.

Despite this explosion in speed, computers aren't able to keep up with the scale of data becoming available. By some estimates, advances in gene sequencing technology will make gene-sequence data available more quickly than processors are getting faster. In other words, for genetic data, computers are become less and less able to cope with the scale of processing problems each year, even though the computers themselves are getting faster.

To circumvent physical and mechanical constraints on individual processor speed, manufacturers are turning to another solution: multiple processors. If two, or three, or more processors are available, then many programs can be executed more quickly. While one processor is doing one aspect of some computation, others can work on another. All of them can share the same data, but the work will proceed in parallel.

In order to be able to work together, multiple processors need to be able to share information with each other. This is accomplished using a shared-memory environment. The variables, objects, and data structures in that environment are accessible to all the processes.The role of a processor in computation is to carry out the evaluation and execution rules of a programming language. In a shared memory model, different processes may execute different statements, but any statement can affect the shared environment.

4.3.1 The Problem with Shared State

Sharing state between multiple processes creates problems that a single-process environments do not have. To understand why, let us look the following simple calculation:

```
x = 5
x = square(x)
x = x + 1
```

The value of x is time-dependent. At first, it is 5, then, some time later, it is 25, and then finally, it is 26. In a single-process environment, this time-dependence is is not a problem. The value of `x` at the end is always 26. The same cannot be said, however, if multiple processes exist. Suppose we executed the last 2 lines of above code in parallel: one

processor executes `x = square(x)` and the other executes `x = x+1`. Each of these assignment statements involves looking up the value currently bound to `x`, then updating that binding with a new value. Let us assume that since `x` is shared, only a single process will read or write it at a time. Even so, the order of the reads and writes may vary. For instance, the example below shows a series of steps for each of two processes, `P1` and `P2`. Each step is a part of the evaluation process described briefly, and time proceeds from top to bottom:

```
P1                    P2
read x: 5
                      read x: 5
calculate 5*5: 25     calculate 5+1: 6
write 25 -> x
                      write x-> 6
```

In this order, the final value of `x` is 6. If we do not coordinate the two processes, we could have another order with a different result:

```
P1                    P2
                      read x: 5
read x: 5             calculate 5+1: 6
calculate 5*5: 25     write x->6
write 25 -> x
```

In this ordering, `x` would be 25. In fact, there are multiple possibilities depending on the order in which the processes execute their lines. The final value of `x` could end up being 5, 25, or the intended value, 26.

The preceding example is trivial. `square(x)` and `x = x + 1` are simple calculations that are fast. We don't lose much time by forcing one to go after the other. But what about situations in which parallelization is essential? An example of such a situation is banking. At any given time, there may be thousands of people wanting to make transactions with their bank accounts: they may want to swipe their cards at shops, deposit checks, transfer money, or pay bills. Even a single account may have multiple transactions active at the same time.

Let us look at how the `make_withdraw` function from Chapter 2, modified below to print the balance after updating it rather than return it. We are interested in how this function will perform in a concurrent situation.

```
>>> def make_withdraw(balance):
        def withdraw(amount):
            nonlocal balance
            if amount > balance:
                print('Insufficient funds')
            else:
                balance = balance - amount
                print(balance)
        return withdraw
```

Now imagine that we create an account with $10 in it. Let us think about what happens if we withdraw too much money from the account. If we do these transactions in order, we receive an insufficient funds message.

```
>>> w = make_withdraw(10)
>>> w(8)
2
>>> w(7)
'Insufficient funds'
```

In parallel, however, there can be many different outcomes. One possibility appears below:

```
P1: w(8)                        P2: w(7)
read balance: 10
read amount: 8                  read balance: 10
8 > 10: False                   read amount: 7
if False                        7 > 10: False
10 - 8: 2                       if False
write balance -> 2              10 - 7: 3
read balance: 2                 write balance -> 3
print 2                         read balance: 3
                                print 3
```

This particular example gives an incorrect outcome of 3. It is as if the `w(8)` transaction never happened! Other possible outcomes are 2, and `'Insufficient funds'`. The source of the problems are the following: if `P2` reads `balance` before `P1` has written to `balance` (or vice versa), `P2`'s state is *inconsistent*. The value of `balance` that `P2` has read is obsolete, and `P1` is going to change it. `P2` doesn't know that and will overwrite it with an inconsistent value.

These example shows that parallelizing code is not as easy as dividing up the lines between multiple processors and having them be executed. The order in which variables are read and written matters.

A tempting way to enforce correctness is to stipulate that no two programs that modify shared data can run at the same time. For banking, unfortunately, this would mean that only one transaction could proceed at a time, since all transactions modify shared data. Intuitively, we understand that there should be no problem allowing 2 different people to perform transactions on completely separate accounts simultaneously. Somehow, those two operations do not interfere with each other the same way that simultaneous operations on the same account interfere with each other. Moreover, there is no harm in letting processes run concurrently when they are not reading or writing.

4.3.2 Correctness in Parallel Computation

There are two criteria for correctness in parallel computation environments. The first is that the outcome should always be the same. The second is that the outcome should be the same as if the code was executed in serial.

The first condition says that we must avoid the variability shown in the previous section, in which interleaving the reads and writes in different ways produces different results. In the example in which we withdrew `w(8)` and `w(7)` from a $10 account, this condition says that we must always return the same answer independent of the order in which `P1` 's and `P2` 's instructions are executed. Somehow, we must write our programs in such a way that, no matter how they are interleaved with each other, they should always produce the same result.

The second condition pins down which of the many possible outcomes is correct. In the example in which we evaluated `w(7)` and `w(8)` from a $10 account, this condition says that the result must always come out to be `Insufficient funds` , and not 2 or 3.

Problems arise in parallel computation when one process influences another during **critical sections** of a program. These are sections of code that need to be executed as if they were a single instruction, but are actually made up of smaller statements. A program's execution is conducted as a series of **atomic** hardware instructions, which are instructions that cannot be broken in to smaller units or interrupted because of the design of the processor. In order to behave correctly in concurrent situations, the critical sections in a programs code need to be have *atomicity* -- a guarantee that they will not be interrupted by any other code.

To enforce the atomicity of critical sections in a program's code under concurrency , there need to be ways to force processes to either *serialize* or *synchronize* with each other at important times. Serialization means that only one process runs at a time -- that they temporarily act as if they were being executed in serial. Synchronization takes two forms. The first is **mutual exclusion**, processes taking turns to access a variable, and the second is **conditional synchronization**, processes waiting until a condition is satisfied (such as other processes having finished their task) before continuing. This way, when one program is about to enter a critical section, the other processes can wait until it finishes, and then proceed safely.

4.3.3 Protecting Shared State: Locks and Semaphores

All the methods for synchronization and serialization that we will discuss in this section use the same underlying idea. They use variables in shared state as *signals* that all the processes understand and respect. This is the same philosophy that allows computers in a

distributed system to work together -- they coordinate with each other by passing messages according to a protocol that every participant understands and respects.

These mechanisms are not physical barriers that come down to protect shared state. Instead they are based on mutual understanding. It is the same sort of mutual understanding that allows traffic going in multiple directions to safely use an intersection. There are no physical walls that stop cars from crashing into each other, just respect for rules that say red means "stop", and green means "go". Similarly, there is really nothing protecting those shared variables except that the processes are programmed only to access them when a particular signal indicates that it is their turn.

Locks. Locks, also known as *mutexes* (short for mutual exclusions), are shared objects that are commonly used to signal that shared state is being read or modified. Different programming languages implement locks in different ways, but in Python, a process can try to acquire "ownership" of a lock using the `acquire()` method, and then `release()` it some time later when it is done using the shared variables. While a lock is acquired by a process, any other process that tries to perform the `acquire()` action will automatically be made to wait until the lock becomes free. This way, only one process can acquire a lock at a time.

For a lock to protect a particular set of variables, all the processes need to be programmed to follow a rule: no process will access any of the shared variables unless it owns that particular lock. In effect, all the processes need to "wrap" their manipulation of the shared variables in `acquire()` and `release()` statements for that lock.

We can apply this concept to the bank balance example. The critical section in that example was the set of operations starting when `balance` was read to when `balance` was written. We saw that problems occurred if more than one process was in this section at the same time. To protect the critical section, we will use a lock. We will call this lock `balance_lock` (although we could call it anything we liked). In order for the lock to actually protect the section, we must make sure to `acquire()` the lock before trying to entering the section, and `release()` the lock afterwards, so that others can have their turn.

```
>>> from threading import Lock
>>> def make_withdraw(balance):
        balance_lock = Lock()
        def withdraw(amount):
            nonlocal balance
            # try to acquire the lock
            balance_lock.acquire()
            # once successful, enter the critical section
            if amount > balance:
                print("Insufficient funds")
            else:
                balance = balance - amount
                print(balance)
            # upon exiting the critical section, release the lock
            balance_lock.release()
```

If we set up the same situation as before:

```
w = make_withdraw(10)
```

And now execute `w(8)` and `w(7)` in parallel:

```
P1                                    P2
acquire balance_lock: ok
read balance: 10                      acquire balance_lock: wait
read amount: 8                        wait
8 > 10: False                         wait
if False                              wait
10 - 8: 2                             wait
write balance -> 2                    wait
read balance: 2                       wait
print 2                               wait
release balance_lock                  wait
                                      acquire balance_lock:ok
                                      read balance: 2
                                      read amount: 7
                                      7 > 2: True
                                      if True
                                      print 'Insufficient funds'
                                      release balance_lock
```

We see that it is impossible for two processes to be in the critical section at the same time. The instant one process acquires balance*lock, the other one has to wait until that processes _finishes* its critical section before it can even start.

Note that the program will not terminate unless `P1` releases `balance_lock`. If it does not release `balance_lock`, `P2` will never be able to acquire it and will be stuck waiting forever. Forgetting to release acquired locks is a common error in parallel programming.

Semaphores. Semaphores are signals used to protect access to limited resources. They are similar to locks, except that they can be acquired multiple times up to a limit. They are like elevators that can only carry a certain number of people. Once the limit has been reached, a process must wait to use the resource until another process releases the semaphore and it can acquire it.

For example, suppose there are many processes that need to read data from a central database server. The server may crash if too many processes access it at once, so it is a good idea to limit the number of connections. If the database can only support *N=2* connections at once, we can set up a semaphore with value *N*=2.

```
>>> from threading import Semaphore
>>> db_semaphore = Semaphore(2) # set up the semaphore
>>> database = []
>>> def insert(data):
        db_semaphore.acquire() # try to acquire the semaphore
        database.append(data)  # if successful, proceed
        db_semaphore.release() # release the semaphore
```

```
>>> insert(7)
>>> insert(8)
>>> insert(9)
```

The semaphore will work as intended if all the processes are programmed to only access the database if they can acquire the semaphore. Once *N=2* processes have acquired the semaphore, any other processes will wait until one of them has released the semaphore, and then try to acquire it before accessing the database:

```
P1                          P2                          P3
acquire db_semaphore: ok    acquire db_semaphore: wait  acquire db_semaphore: ok
read data: 7                wait                        read data: 9
append 7 to database        wait                        append 9 to database
release db_semaphore: ok    acquire db_semaphore: ok    release db_semaphore: ok
                            read data: 8
                            append 8 to database
                            release db_semaphore: ok
```

A semaphore with value 1 behaves like a lock.

4.3.4 Staying Synchronized: Condition variables

Condition variables are useful when a parallel computation is composed of a series of steps. A process can use a condition variable to signal it has finished its particular step. Then, the other processes that were waiting for the signal can start their work. An example of a computation that needs to proceed in steps a sequence of large-scale vector computations. In computational biology, web-scale computations, and image processing and graphics, it is common to have very large (million-element) vectors and matrices. Imagine the following computation:

$$\begin{aligned} A &= B + C \\ V &= MA \end{aligned}$$

We may choose to parallelize each step by breaking up the matrices and vectors into range of rows, and assigning each range to a separate thread. As an example of the above computation, imagine the following simple values:

$$B = \begin{pmatrix} 2 \\ 0 \end{pmatrix} C = \begin{pmatrix} 0 \\ 5 \end{pmatrix} M = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$$

We will assign first half (in this case the first row) to one thread, and the second half (second row) to another thread:

$$\begin{aligned} P1: A_1 &= B_1 + C_1 \\ P1: V_1 &= M_1 \cdot A \\ P2: A_2 &= B_2 + C_2 \\ P2: V_2 &= M_2 \cdot A \end{aligned}$$

In pseudocode, the computation is:

```
def do_step_1(index):
  A[index] = B[index] + C[index]

def do_step_2(index):
        V[index] = M[index] . A
```

Process 1 does:

```
do_step_1(1)
do_step_2(1)
```

And process 2 does:

```
do_step_1(2)
do_step_2(2)
```

If allowed to proceed without synchronization, the following inconsistencies could result:

```
P1                           P2
read B1: 2
read C1: 0
calculate 2+0: 2
write 2 -> A1                read B2: 0
read M1: (1 2)               read C2: 5
read A: (2 0)                calculate 5+0: 5
calculate (1 2).(2 0): 2     write 5 -> A2
write 2 -> V1                read M2: (1 2)
                             read A: (2 5)
                             calculate (1 2).(2 5):12
                             write 12 -> V2
```

The problem is that V should not be computed until all the elements of A have been computed. However, P1 finishes `A = B+C` and moves on to `V = MA` before all the elements of A have been computed. It therefore uses an inconsistent value of A when multiplying by M.

We can use a condition variable to solve this problem.

Condition variables are objects that act as signals that a condition has been satisfied. They are commonly used to coordinate processes that need to wait for something to happen before continuing. Processes that need the condition to be satisfied can make themselves wait on a condition variable until some other process modifies it to tell them to proceed.

In Python, any number of processes can signal that they are waiting for a condition using the `condition.wait()` method. After calling this method, they automatically wait until some other process calls the `condition.notify()` or `condition.notifyAll()` function. The `notify()` method wakes up just one process, and leaves the others waiting. The `notifyAll()` method wakes up all the waiting processes. Each of these is useful in different situations.

Since condition variables are usually associated with shared variables that determine whether or not the condition is true, they are offer `acquire()` and `release()` methods. These methods should be used when modifying variables that could change the status of the condition. Any process wishing to signal a change in the condition must first get access to it using `acquire()`.

In our example, the condition that must be met before advancing to the second step is that both processes must have finished the first step. We can keep track of the number of processes that have finished a step, and whether or not the condition has been met, by introducing the following 2 variables:

```
step1_finished = 0
start_step2 = Condition()
```

We will insert a `start_step_2().wait()` at the beginning of `do_step_2`. Each process will increment `step1_finished` when it finishes Step 1, but we will only signal the condition when `step_1_finished = 2`. The following pseudocode illustrates this:

```
step1_finished = 0
start_step2 = Condition()

def do_step_1(index):
  A[index] = B[index] + C[index]
  # access the shared state that determines the condition status
  start_step2.acquire()
  step1_finished += 1
  if(step1_finished == 2): # if the condition is met
        start_step2.notifyAll() # send the signal
  #release access to shared state
  start_step2.release()

def do_step_2(index):
  # wait for the condition
  start_step2.wait()
  V[index] = M[index] . A
```

With the introduction of this condition, both processes enter Step 2 together as follows::

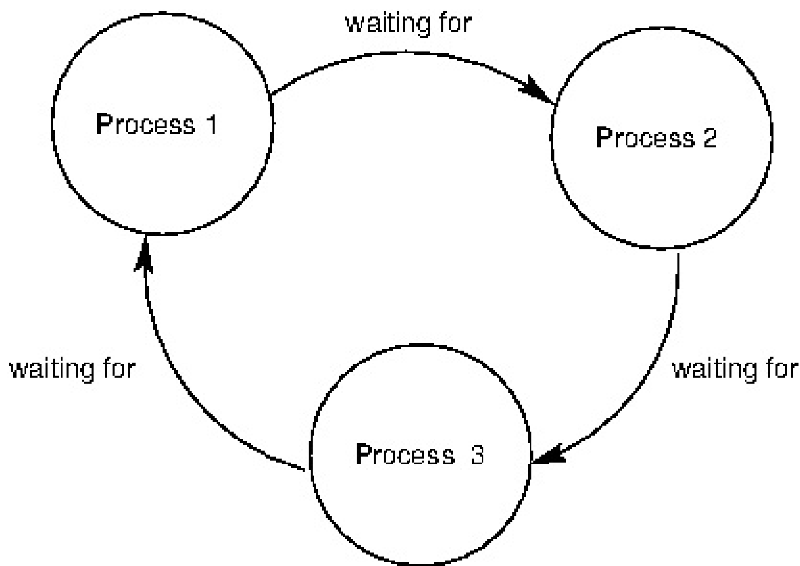
```
P1                              P2
read B1: 2
read C1: 0
calculate 2+0: 2
write 2 -> A1                   read B2: 0
acquire start_step2: ok         read C2: 5
write 1 -> step1_finished       calculate 5+0: 5
step1_finished == 2: false      write 5-> A2
release start_step2: ok         acquire start_step2: ok
start_step2: wait               write 2-> step1_finished
wait                            step1_finished == 2: true
wait                            notifyAll start_step_2: ok
start_step2: ok                 start_step2:ok
read M1: (1 2)                  read M2: (1 2)
read A:(2 5)
calculate (1 2). (2 5): 12      read A:(2 5)
write 12->V1                    calculate (1 2). (2 5): 12
                                write 12->V2
```

Upon entering `do_step_2`, P1 has to wait on `start_step_2` until P2 increments `step1_finished`, finds that it equals 2, and signals the condition.

4.3.5 Deadlock

While synchronization methods are effective for protecting shared state, they come with a catch. Because they cause processes to wait on each other, they are vulnerable to **deadlock**, a situation in which two or more processes are stuck, waiting for each other to finish. We have already mentioned how forgetting to release a lock can cause a process to get stuck indefinitely. But even if there are the correct number of `acquire()` and `release()` calls, programs can still reach deadlock.

The source of deadlock is a **circular wait**, illustrated below. No process can continue because it is waiting for other processes that are waiting for it to complete.



As an example, we will set up a deadlock with two processes. Suppose there are two locks, `x_lock` and `y_lock`, and they are used as follows:

```
>>> x_lock = Lock()
>>> y_lock = Lock()
>>> x = 1
>>> y = 0
>>> def compute():
        x_lock.acquire()
        y_lock.acquire()
        y = x + y
        x = x * x
        y_lock.release()
        x_lock.release()
```

```
>>> def anti_compute():
        y_lock.acquire()
        x_lock.acquire()
        y = y - x
        x = sqrt(x)
        x_lock.release()
        y_lock.release()
```

If `compute()` and `anti_compute()` are executed in parallel, and happen to interleave with each other as follows:

```
P1                          P2
acquire x_lock: ok          acquire y_lock: ok
acquire y_lock: wait        acquire x_lock: wait
wait                        wait
wait                        wait
wait                        wait
...                         ...
```

the resulting situation is a deadlock. `P1` and `P2` are each holding on to one lock, but they need both in order to proceed. `P1` is waiting for `P2` to release `y_lock`, and `P2` is waiting for `P1` to release `x_lock`. As a result, neither can proceed.

Chapter 5: Sequences and Coroutines

Contents

5.1 Introduction

In this chapter, we continue our discussion of real-world applications by developing new tools to process sequential data. In Chapter 2, we introduced a sequence interface, implemented in Python by built-in data types such as `tuple` and `list`. Sequences supported two operations: querying their length and accessing an element by index. In Chapter 3, we developed a user-defined implementations of the sequence interface, the `Rlist` class for representing recursive lists. These sequence types proved effective for representing and accessing a wide variety of sequential datasets.

However, representing sequential data using the sequence abstraction has two important limitations. The first is that a sequence of length *n* typically takes up an amount of memory proportional to *n*. Therefore, the longer a sequence is, the more memory it takes to represent it.

The second limitation of sequences is that sequences can only represent datasets of known, finite length. Many sequential collections that we may want to represent do not have a well-defined length, and some are even infinite. Two mathematical examples of infinite sequences are the positive integers and the Fibonacci numbers. Sequential data sets of unbounded length also appear in other computational domains. For instance, the sequence of all Twitter posts grows longer with every second and therefore does not have a fixed length. Likewise, the sequence of telephone calls sent through a cell tower, the sequence of mouse movements made by a computer user, and the sequence of acceleration measurements from sensors on an aircraft all extend without bound as the world evolves.

In this chapter, we introduce new constructs for working with sequential data that are designed to accommodate collections of unknown or unbounded length, while using limited memory. We also discuss how these tools can be used with a programming construct called a coroutine to create efficient, modular data processing pipelines.

5.2 Implicit Sequences

The central observation that will lead us to efficient processing of sequential data is that a sequence can be *represented* using programming constructs without each element being *stored* explicitly in the memory of the computer. To put this idea into practice, we will construct objects that provides access to all of the elements of some sequential dataset that an application may desire, but without computing all of those elements in advance and storing them.

A simple example of this idea arises in the `range` sequence type introduced in Chapter 2. A `range` represents a consecutive, bounded sequence of integers. However, it is not the case that each element of that sequence is represented explicitly in memory. Instead, when an element is requested from a `range`, it is computed. Hence, we can represent very large ranges of integers without using large blocks of memory. Only the end points of the range are stored as part of the `range` object, and elements are computed on the fly.

```
>>> r = range(10000, 1000000000)
>>> r[45006230]
45016230
```

In this example, not all 999,990,000 integers in this range are stored when the range instance is constructed. Instead, the range object adds the first element 10,000 to the index 45,006,230 to produce the element 45,016,230. Computing values on demand, rather than retrieving them from an existing representation, is an example of *lazy* computation. Computer science is a discipline that celebrates laziness as an important computational tool.

An *iterator* is an object that provides sequential access to an underlying sequential dataset. Iterators are built-in objects in many programming languages, including Python. The iterator abstraction has two components: a mechanism for retrieving the *next* element in some underlying series of elements and a mechanism for signaling that the end of the series has been reached and no further elements remain. In programming languages with built-in object systems, this abstraction typically corresponds to a particular interface that can be implemented by classes. The Python interface for iterators is described in the next section.

The usefulness of iterators is derived from the fact that the underlying series of data for an iterator may not be represented explicitly in memory. An iterator provides a mechanism for considering each of a series of values in turn, but all of those elements do not need to be stored simultaneously. Instead, when the next element is requested from an iterator, that element may be computed on demand instead of being retrieved from an existing memory source.

Ranges are able to compute the elements of a sequence lazily because the sequence represented is uniform, and any element is easy to compute from the starting and ending bounds of the range. Iterators allow for lazy generation of a much broader class of underlying sequential datasets, because they do not need to provide access to arbitrary elements of the underlying series. Instead, they must only compute the next element of the series, in order, each time another element is requested. While not as flexible as accessing arbitrary elements of a sequence (called *random access*), *sequential access* to sequential data series is often sufficient for data processing applications.

5.2.1 Python Iterators

The Python iterator interface includes two messages. The `__next__` message queries the iterator for the next element of the underlying series that it represents. In response to invoking `__next__` as a method, an iterator can perform arbitrary computation in order to either retrieve or compute the next element in an underlying series. Calls to `__next__` make a mutating change to the iterator: they advance the position of the iterator. Hence, multiple calls to `__next__` will return sequential elements of an underlying series. Python signals that the end of an underlying series has been reached by raising a `StopIteration` exception during a call to `__next__`.

The `Letters` class below iterates over an underlying series of letters from `a` to `d`. The member variable `current` stores the current letter in the series, and the `__next__` method returns this letter and uses it to compute a new value for `current`.

```
>>> class Letters(object):
        def __init__(self):
            self.current = 'a'
        def __next__(self):
            if self.current > 'd':
                raise StopIteration
            result = self.current
            self.current = chr(ord(result)+1)
            return result
        def __iter__(self):
            return self
```

The `__iter__` message is the second required message of the Python iterator interface. It simply returns the iterator; it is useful for providing a common interface to iterators and sequences, as described in the next section.

Using this class, we can access letters in sequence.

```
>>> letters = Letters()
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```

A `Letters` instance can only be iterated through once. Once its `__next__()` method raises a `StopIteration` exception, it continues to do so from then on. There is no way to reset it; one must create a new instance.

Iterators also allow us to represent infinite series by implementing a `__next__` method that never raises a `StopIteration` exception. For example, the `Positives` class below iterates over the infinite series of positive integers.

```
>>> class Positives(object):
        def __init__(self):
            self.current = 0;
        def __next__(self):
            result = self.current
            self.current += 1
            return result
        def __iter__(self):
            return self
```

5.2.2 For Statements

In Python, sequences can expose themselves to iteration by implementing the `__iter__` message. If an object represents sequential data, it can serve as an *iterable* object in a `for` statement by returning an iterator object in response to the `__iter__` message. This iterator is meant to have a `__next__()` method that returns each element of the sequence in turn, eventually raising a `StopIteration` exception when the end of the sequence is reached.

```
>>> counts = [1, 2, 3]
>>> for item in counts:
        print(item)
1
2
3
```

In the above example, the `counts` list returns an iterator in response to a call to its `__iter__()` method. The `for` statement then calls that iterator's `__next__()` method repeatedly, and assigns the returned value to `item` each time. This process continues until

the iterator raises a `StopIteration` exception, at which point the `for` statement concludes.

With our knowledge of iterators, we can implement the evaluation rule of a `for` statement in terms of `while`, assignment, and `try` statements.

```
>>> i = counts.__iter__()
>>> try:
        while True:
            item = i.__next__()
            print(item)
    except StopIteration:
        pass
1
2
3
```

Above, the iterator returned by invoking the `__iter__` method of `counts` is bound to a name `i` so that it can be queried for each element in turn. The handling clause for the `StopIteration` exception does nothing, but handling the exception provides a control mechanism for exiting the `while` loop.

5.2.3 Generators and Yield Statements

The `Letters` and `Positives` objects above require us to introduce a new field `self.current` into our object to keep track of progress through the sequence. With simple sequences like those shown above, this can be done easily. With complex sequences, however, it can be quite difficult for the `__next__()` function to save its place in the calculation. Generators allow us to define more complicated iterations by leveraging the features of the Python interpreter.

A *generator* is an iterator returned by a special class of function called a *generator function*. Generator functions are distinguished from regular functions in that rather than containing `return` statements in their body, they use `yield` statement to return elements of a series.

Generators do not use attributes of an object to track their progress through a series. Instead, they control the execution of the generator function, which runs until the next `yield` statement is executed each time the generator's `__next__` method is invoked. The `Letters` iterator can be implemented much more compactly using a generator function.

```
>>> def letters_generator():
        current = 'a'
        while current <= 'd':
            yield current
            current = chr(ord(current)+1)
```

```
>>> for letter in letters_generator():
        print(letter)
a
b
c
d
```

Even though we never explicitly defined `__iter__()` or `__next__()` methods, Python understands that when we use the `yield` statement, we are defining a generator function. When called, a generator function doesn't return a particular yielded value, but instead a `generator` (which is a type of iterator) that itself can return the yielded values. A generator object has `__iter__` and `__next__` methods, and each call to `__next__` continues execution of the generator function from wherever it left off previously until another `yield` statement is executed.

The first time `__next__` is called, the program executes statements from the body of the `letters_generator` function until it encounters the `yield` statement. Then, it pauses and returns the value of `current`. `yield` statements do not destroy the newly created environment, they preserve it for later. When `__next__` is called again, execution resumes where it left off. The values of `current` and of any other bound names in the scope of `letters_generator` are preserved across subsequent calls to `__next__`.

We can walk through the generator by manually calling `____next__()`:

```
>>> letters = letters_generator()
>>> type(letters)
<class 'generator'>
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The generator does not start executing any of the body statements of its generator function until the first time `__next__()` is called.

5.2.4 Iterables

In Python, iterators only make a single pass over the elements of an underlying series. After that pass, the iterator will continue to raise a `StopIteration` exception when `__next__()` is called. Many applications require iteration over elements multiple times. For example, we have to iterate over a list many times in order to enumerate all pairs of elements.

```
>>> def all_pairs(s):
        for item1 in s:
            for item2 in s:
                yield (item1, item2)
```

```
>>> list(all_pairs([1, 2, 3]))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

Sequences are not themselves iterators, but instead *iterable* objects. The iterable interface in Python consists of a single message, `__iter__`, that returns an iterator. The built-in sequence types in Python return new instances of iterators when their `__iter__` methods are invoked. If an iterable object returns a fresh instance of an iterator each time `__iter__` is called, then it can be iterated over multiple times.

New iterable classes can be defined by implementing the iterable interface. For example, the *iterable* `LetterIterable` class below returns a new iterator over letters each time `__iter__` is invoked.

```
>>> class LetterIterable(object):
        def __iter__(self):
            current = 'a'
            while current <= 'd':
                yield current
                current = chr(ord(current)+1)
```

The `__iter__` method is a generator function; it returns a generator object that yields the letters `'a'` through `'d'`.

A `Letters` iterator object gets "used up" after a single iteration, whereas the `LetterIterable` object can be iterated over multiple times. As a result, a `LetterIterable` instance can serve as an argument to `all_pairs`.

```
>>> letters = LetterIterable()
>>> all_pairs(letters).__next__()
('a', 'a')
```

5.2.5 Streams

Streams offer a final way to represent sequential data implicity. A stream is a lazily computed recursive list. Like the `Rlist` class from Chapter 3, a `Stream` instance responds to requests for its `first` element and the `rest` of the stream. Like an `Rlist`, the `rest` of a `Stream` is itself a `Stream`. Unlike an `Rlist`, the `rest` of a stream is only computed when it is looked up, rather than being stored in advance. That is, the `rest` of a stream is computed lazily.

To achieve this lazy evaluation, a stream stores a function that computes the rest of the stream. Whenever this function is called, its returned value is cached as part of the stream in an attribute called `_rest`, named with an underscore to indicate that it should not be accessed directly. The accessible attribute `rest` is a property method that returns the rest of the stream, computing it if necessary. With this design, a stream stores *how to compute* the rest of the stream, rather than always storing it explicitly.

```
>>> class Stream(object):
        """A lazily computed recursive list."""
        def __init__(self, first, compute_rest, empty=False):
            self.first = first
            self._compute_rest = compute_rest
            self.empty = empty
            self._rest = None
            self._computed = False
        @property
        def rest(self):
            """Return the rest of the stream, computing it if necessary."""
            assert not self.empty, 'Empty streams have no rest.'
            if not self._computed:
                self._rest = self._compute_rest()
                self._computed = True
            return self._rest
        def __repr__(self):
            if self.empty:
                return '<empty stream>'
            return 'Stream({0}, <compute_rest>)'.format(repr(self.first))
```

```
>>> Stream.empty = Stream(None, None, True)
```

A recursive list is defined using a nested expression. For example, we can create an `Rlist` that represents the elements `1` then `5` as follows:

```
>>> r = Rlist(1, Rlist(2+3, Rlist.empty))
```

Likewise, we can create a `Stream` representing the same series. The `Stream` does not actually compute the second element `5` until the rest of the stream is requested.

```
>>> s = Stream(1, lambda: Stream(2+3, lambda: Stream.empty))
```

Here, `1` is the first element of the stream, and the `lambda` expression that follows returns a function for computing the rest of the stream. The second element of the computed stream is a function that returns an empty stream.

Accessing the elements of recursive list `r` and stream `s` proceed similarly. However, while `5` is stored within `r`, it is computed on demand for `s` via addition the first time that it is requested.

```
>>> r.first
1
>>> s.first
1
>>> r.rest.first
5
>>> s.rest.first
5
>>> r.rest
Rlist(5)
>>> s.rest
Stream(5, <compute_rest>)
```

While the `rest` of `r` is a one-element recursive list, the `rest` of `s` includes a function to compute the rest; the fact that it will return the empty stream may not yet have been discovered.

When a `Stream` instance is constructed, the field `self._computed` is `False`, signifying that the `_rest` of the `Stream` has not yet been computed. When the `rest` attribute is requested via a dot expression, the `rest` method is invoked, which triggers computation with `self._rest = self.compute_rest`. Because of the caching mechanism within a `Stream`, the `compute_rest` function is only ever called once.

The essential properties of a `compute_rest` function are that it takes no arguments, and it returns a `Stream`.

Lazy evaluation gives us the ability to represent infinite sequential datasets using streams. For example, we can represent increasing integers, starting at any `first` value.

```
>>> def make_integer_stream(first=1):
      def compute_rest():
        return make_integer_stream(first+1)
      return Stream(first, compute_rest)
```

```
>>> ints = make_integer_stream()
>>> ints
Stream(1, <compute_rest>)
>>> ints.first
1
```

When `make_integer_stream` is called for the first time, it returns a stream whose `first` is the first integer in the sequence (`1` by default). However, `make_integer_stream` is actually recursive because this stream's `compute_rest` calls `make_integer_stream` again, with an incremented argument. This makes `make_integer_stream` recursive, but also lazy.

```
>>> ints.first
1
>>> ints.rest.first
2
>>> ints.rest.rest
Stream(3, <compute_rest>)
```

Recursive calls are only made to `make_integer_stream` whenever the `rest` of an integer stream is requested.

The same higher-order functions that manipulate sequences -- `map` and `filter` -- also apply to streams, although their implementations must change to apply their argument functions lazily. The function `map_stream` maps a function over a stream, which produces a new stream. The locally defined `compute_rest` function ensures that the function will be mapped onto the rest of the stream whenever the rest is computed.

```
>>> def map_stream(fn, s):
        if s.empty:
            return s
        def compute_rest():
            return map_stream(fn, s.rest)
        return Stream(fn(s.first), compute_rest)
```

A stream can be filtered by defining a `compute_rest` function that applies the filter function to the rest of the stream. If the filter function rejects the first element of the stream, the rest is computed immediately. Because `filter_stream` is recursive, the rest may be computed multiple times until a valid `first` element is found.

```
>>> def filter_stream(fn, s):
        if s.empty:
            return s
        def compute_rest():
            return filter_stream(fn, s.rest)
        if fn(s.first):
            return Stream(s.first, compute_rest)
        return compute_rest()
```

The `map_stream` and `filter_stream` functions exhibit a common pattern in stream processing: a locally defined `compute_rest` function recursively applies a processing function to the rest of the stream whenever the rest is computed.

To inspect the contents of a stream, we can truncate it to finite length and convert it to a Python `list`.

```
>>> def truncate_stream(s, k):
        if s.empty or k == 0:
            return Stream.empty
        def compute_rest():
            return truncate_stream(s.rest, k-1)
        return Stream(s.first, compute_rest)
```

```
>>> def stream_to_list(s):
        r = []
        while not s.empty:
            r.append(s.first)
            s = s.rest
        return r
```

These convenience functions allow us to verify our `map_stream` implementation with a simple example that squares the integers from `3` to `7`.

```
>>> s = make_integer_stream(3)
>>> s
Stream(3, <compute_rest>)
>>> m = map_stream(lambda x: x*x, s)
>>> m
Stream(9, <compute_rest>)
>>> stream_to_list(truncate_stream(m, 5))
[9, 16, 25, 36, 49]
```

We can use our `filter_stream` function to define a stream of prime numbers using the sieve of Eratosthenes, which filters a stream of integers to remove all numbers that are multiples of its first element. By successively filtering with each prime, all composite numbers are removed from the stream.

```
>>> def primes(pos_stream):
        def not_divible(x):
            return x % pos_stream.first != 0
        def compute_rest():
            return primes(filter_stream(not_divible, pos_stream.rest))
        return Stream(pos_stream.first, compute_rest)
```

By truncating the `primes` stream, we can enumerate any prefix of the prime numbers.

```
>>> p1 = primes(make_integer_stream(2))
>>> stream_to_list(truncate_stream(p1, 7))
[2, 3, 5, 7, 11, 13, 17]
```

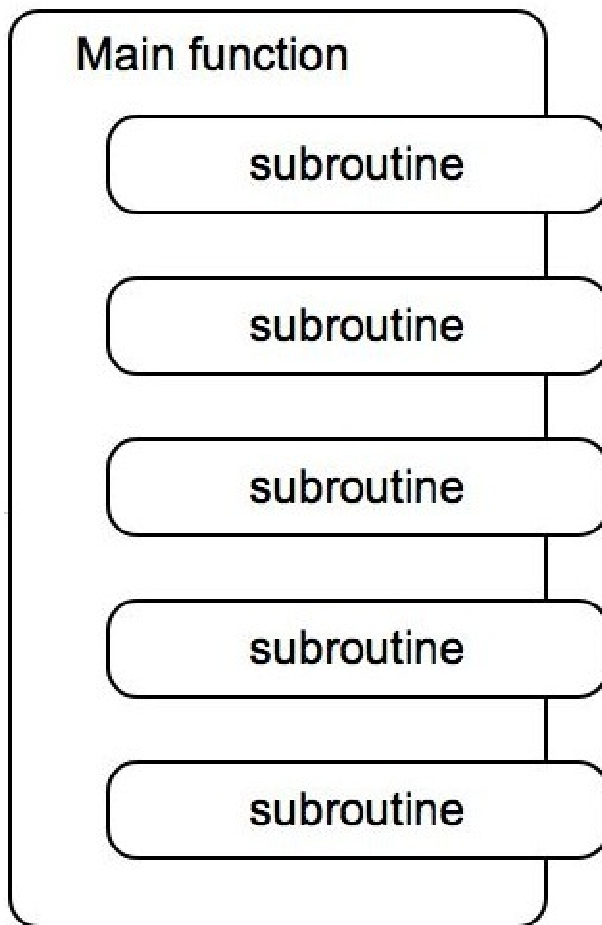
Streams contrast with iterators in that they can be passed to pure functions multiple times and yield the same result each time. The primes stream is not "used up" by converting it to a list. That is, the `first` element of `p1` is still `2` after converting the prefix of the stream to a list.

```
>>> p1.first
2
```

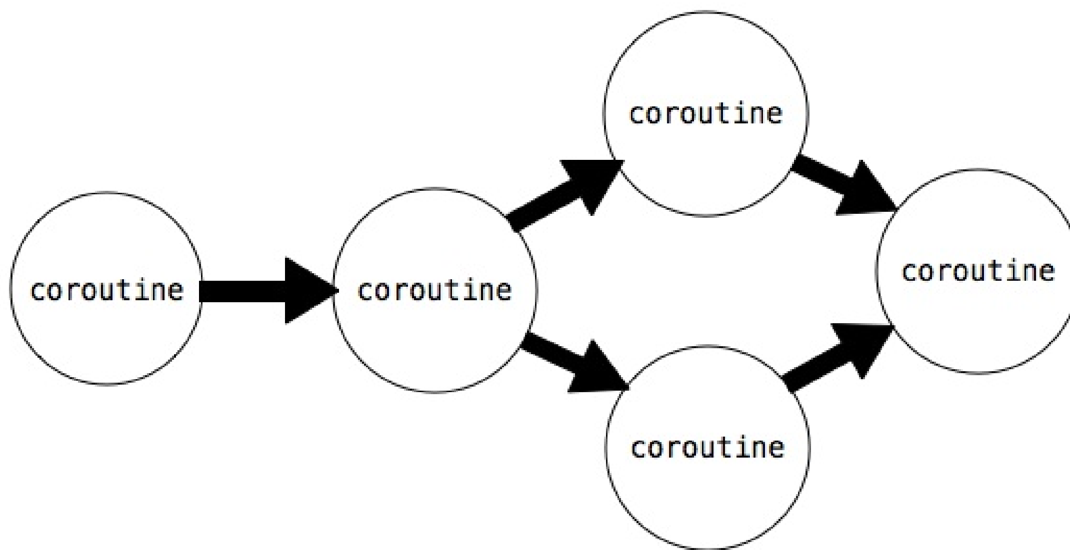
Just as recursive lists provide a simple implementation of the sequence abstraction, streams provide a simple, functional, recursive data structure that implements lazy evaluation through the use of higher-order functions.

5.3 Coroutines

Much of this text has focused on techniques for decomposing complex programs into small, modular components. When the logic for a function with complex behavior is divided into several self-contained steps that are themselves functions, these functions are called helper functions or *subroutines*. Subroutines are called by a main function that is responsible for coordinating the use of several subroutines.



In this section, we introduce a different way of decomposing complex computations using *coroutines*, an approach that is particularly applicable to the task of processing sequential data. Like a subroutine, a coroutine computes a single step of a complex computation. However, when using coroutines, there is no main function to coordinate results. Instead coroutines themselves link together to form a pipeline. There may be a coroutine for consuming the incoming data and sending it to other coroutines. There may be coroutines that each do simple processing steps on data sent to them, and there may finally be another coroutine that outputs a final result.



The difference between coroutines and subroutines is conceptual: subroutines slot into an overarching function to which they are subordinate, whereas coroutines are all colleagues, they cooperate to form a pipeline without any supervising function responsible for calling them in a particular order.

In this section, we will learn how Python supports building coroutines with the `yield` and `send()` statements. Then, we will look at different roles that coroutines can play in a pipeline, and how coroutines can support multitasking.

5.3.1 Python Coroutines

In the previous section, we introduced generator functions, which use `yield` to return values. Python generator functions can also consume values using a `(yield)` statement. In addition two new methods on generator objects, `send()` and `close()`, create a framework for objects that *consume* and produce values. Generator functions that define these objects are coroutines.

Coroutines consume values using a `(yield)` statement as follows:

```
value = (yield)
```

With this syntax, execution pauses at this statement until the object's `send` method is invoked with an argument:

```
coroutine.send(data)
```

Then, execution resumes, with `value` being assigned to the value of `data`. To signal the end of a computation, we shut down a coroutine using the `close()` method. This raises a `GeneratorExit` exception inside the coroutine, which we can catch with a `try/except` clause.

The example below illustrates these concepts. It is a coroutine that prints strings that match a provided pattern.

```
>>> def match(pattern):
        print('Looking for ' + pattern)
        try:
            while True:
                s = (yield)
                if pattern in s:
                    print(s)
        except GeneratorExit:
            print("=== Done ===")
```

We initialize it with a pattern, and call `__next__()` to start execution:

```
>>> m = match("Jabberwock")
>>> m.__next__()
Looking for Jabberwock
```

The call to `__next__()` causes the body of the function to be executed, so the line "Looking for jabberwock" gets printed out. Execution continues until the statement `line = (yield)` is encountered. Then, execution pauses, and waits for a value to be sent to `m`. We can send values to it using `send`.

```
>>> m.send("the Jabberwock with eyes of flame")
the Jabberwock with eyes of flame
>>> m.send("came whiffling through the tulgey wood")
>>> m.send("and burbled as it came")
>>> m.close()
=== Done ===
```

When we call `m.send` with a value, evaluation resumes inside the coroutine `m` at the statement `line = (yield)`, where the sent value is assigned to the variable `line`. Evaluation continues inside `m`, printing out the line if it matches, going through the loop until it encounters `line = (yield)` again. Then, evaluation pauses inside `m` and resumes where `m.send` was called.

We can chain functions that `send()` and functions that `yield` together achieve complex behaviors. For example, the function below splits a string named `text` into words and sends each word to another coroutine.

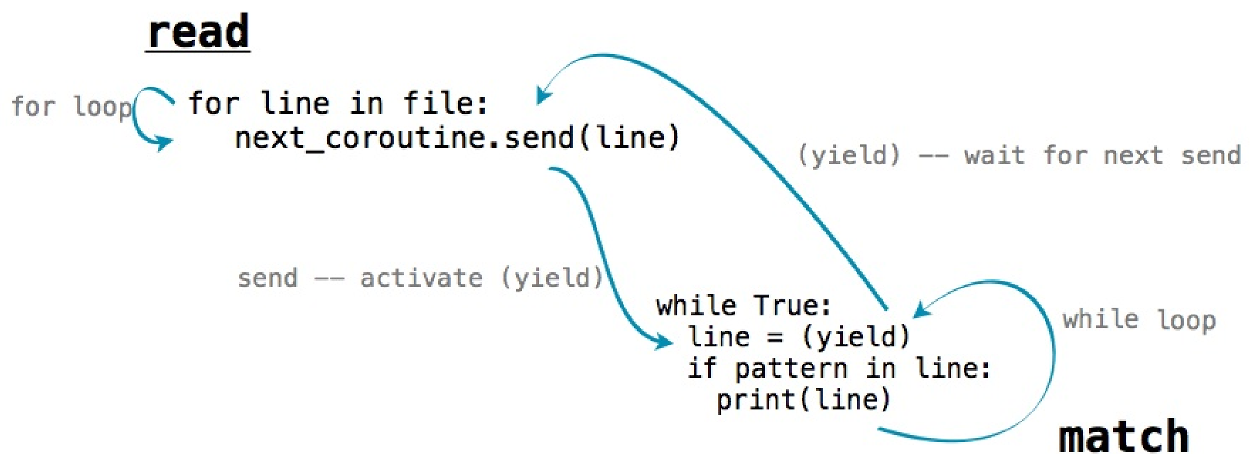
```
>>> def read(text, next_coroutine):
        for line in text.split():
            next_coroutine.send(line)
        next_coroutine.close()
```

Each word is sent to the coroutine bound to `next_coroutine`, causing `next_coroutine` to start executing, and this function to pause and wait. It waits until `next_coroutine` pauses, at which point the function resumes by sending the next word or completing.

If we chain this function together with `match` defined above, we can create a program that prints out only the words that match a particular word.

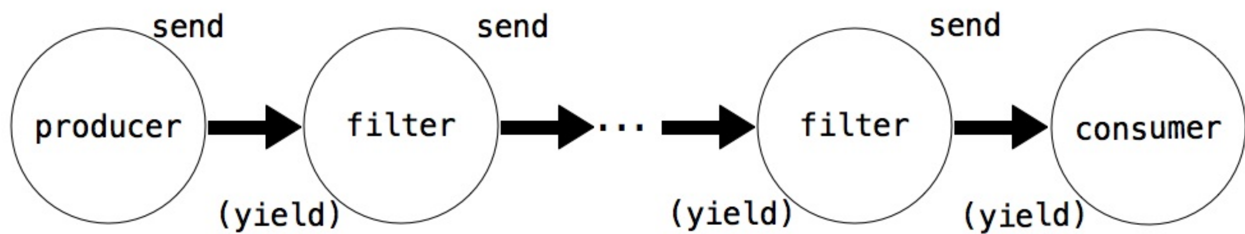
```
>>> text = 'Commending spending is offending to people pending lending!'
>>> matcher = match('ending')
>>> matcher.__next__()
Looking for ending
>>> read(text, matcher)
Commending
spending
offending
pending
lending!
=== Done ===
```

The `read` function sends each word to the coroutine `matcher`, which prints out any input that matches its `pattern`. Within the `matcher` coroutine, the line `s = (yield)` waits for each sent word, and it transfers control back to `read` when it is reached.



5.3.2 Produce, Filter, and Consume

Coroutines can have different roles depending on how they use `yield` and `send()`:



- A **Producer** creates items in a series and uses send(), but not `(yield)`
- A **Filter** uses `(yield)` to consume items and `send()` to send result to a next step.
- A **Consumer** uses `(yield)` to consume items, but does not send.

The function `read` above is an example of a *producer*. It does not use `(yield)`, but uses `send` to produce data items. The function `match` is an example of a consumer. It does not `send` anything, but consumes data with `(yield)`.We can break up `match` into a filter and a consumer. The filter would be a coroutine that only sends on strings that match its pattern.

```
>>> def match_filter(pattern, next_coroutine):
        print('Looking for ' + pattern)
        try:
            while True:
                s = (yield)
                if pattern in s:
                    next_coroutine.send(s)
        except GeneratorExit:
            next_coroutine.close()
```

And the consumer would be a function that printed out lines sent to it.

```
>>> def print_consumer():
        print('Preparing to print')
        try:
            while True:
                line = (yield)
                print(line)
        except GeneratorExit:
            print("=== Done ===")
```

When a filter or consumer is constructed, its `__next__` method must be invoked to start its execution.

```
>>> printer = print_consumer()
>>> printer.__next__()
Preparing to print
>>> matcher = match_filter('pend', printer)
>>> matcher.__next__()
Looking for pend
>>> read(text, matcher)
spending
pending
=== Done ===
```

Even though the name *filter* implies removing items, filters can transform items as well. The function below is an example of a filter that transforms items. It consumes strings and sends along a dictionary of the number of times different letters occur in the string.

```
>>> def count_letters(next_coroutine):
        try:
            while True:
                s = (yield)
                counts = {letter:s.count(letter) for letter in set(s)}
                next_coroutine.send(counts)
        except GeneratorExit as e:
            next_coroutine.close()
```

We can use it to count the most frequently-used letters in text using a consumer that adds up dictionaries and finds the most frequent key.

```
>>> def sum_dictionaries():
        total = {}
        try:
            while True:
                counts = (yield)
                for letter, count in counts.items():
                    total[letter] = count + total.get(letter, 0)
        except GeneratorExit:
            max_letter = max(total.items(), key=lambda t: t[1])[0]
            print("Most frequent letter: " + max_letter)
```

To run this pipeline on a file, we must first read the lines of a file one-by-one. Then, we send the results through `count_letters` and finally to `sum_dictionaries`. We can re-use the `read` coroutine to read the lines of a file.

```
>>> s = sum_dictionaries()
>>> s.__next__()
>>> c = count_letters(s)
>>> c.__next__()
>>> read(text, c)
Most frequent letter: n
```

5.3.3 Multitasking

A producer or filter does not have to be restricted to just one next step. It can have multiple coroutines downstream of it, and `send()` data to all of them. For example, here is a version of `read` that sends the words in a string to multiple next steps.

```
>>> def read_to_many(text, coroutines):
        for word in text.split():
            for coroutine in coroutines:
                coroutine.send(word)
        for coroutine in coroutines:
            coroutine.close()
```

We can use it to examine the same text for multiple words:

```
>>> m = match("mend")
>>> m.__next__()
Looking for mend
>>> p = match("pe")
>>> p.__next__()
Looking for pe
>>> read_to_many(text, [m, p])
Commending
spending
people
pending
=== Done ===
=== Done ===
```

First, `read_to_many` calls `send(word)` on `m`. The coroutine, which is waiting at `text = (yield)` runs through its loop, prints out a match if found, and resumes waiting for the next `send`. Execution then returns to `read_to_many`, which proceeds to send the same line to `p`. Thus, the words of `text` are printed in order.